Standard Compliant Snapshotting for SystemC Virtual Platforms

Von der
Carl-Friedrich-Gauß-Fakultät
der Technischen Universität Carolo-Wilhelmina zu Braunschweig

zur Erlangung des Grades eines
Doktoringenieurs (Dr.-Ing.)

genehmigte Dissertation

von
Bastian Farkas
geboren am 17.09.1984
in Hildesheim

Eingereicht am:	11. April 2019
Disputation am:	24. September 2019
1. Referentin/Referent:	Prof. Dr.-Ing. Mladen Berekovic
2. Referentin/Referent:	Prof. Dr.-Ing. Harald Michalik

2019

Kurzfassung

Die stetige Steigerung der Komplexität eingebetteter Systeme geht einher mit einer ebenso steigenden Komplexität des Entwurfsprozesses. Der Design-Prozess elektronischer Systeme hat mit simplen Schaltkreisentwürfen begonnen. Als die Schaltkreise zu komplex wurden, folgte die Einführung von Hardware-Beschreibungssprachen, die es ermöglichten, textuelle Beschreibungen von Funktionen in Schaltkreis-Designs zu übersetzen. Nach einer Weile erreichte auch das Entwerfen vollständiger Systeme mit Hilfe von Hardware-Beschreibungssprachen einen Grad an Komplexität, der die Benutzung von Beschreibungssprachen nicht mehr sinnvoll erscheinen ließ. Also wurde eine weitere Abstraktionsebene eingeführt: Modellierung auf Transaktionsebene mit SystemC [1][2]. Wir befinden uns momentan in der Übergangsphase vom Entwurf von eingebetteten Systemen basierend auf Hardware-Beschreibungssprachen hin zum Entwurf ebendieser basierend auf virtuellen Plattformen. Die Entwicklungskomplexität hat sich zwischen den Jahren 2003 und 2013 verdreifacht [3]. Da die Entwurfskomplexität rasanter steigt als die Produktivität der Entwickler, entsteht eine Kluft. Diese Dissonanz wurde in 2007 von der *International Technology Roadmap for Semiconductors* als Entwurfskluft (engl. *design gap*) beschrieben [4].

Dieser Zuwachs an Komplexität hat in den letzten fünf Jahren nicht abgenommen. Somit hat sich die Entwurfskluft vergrößert. Die Produktivität wiederherzustellen und gleichzeitig die gesteigerte Entwurfskomplexität zu bewältigen, kann auch erreicht werden, indem der Fokus auf die Entwicklung neuer Werkzeuge und Entwurfsmethoden gelegt wird.

In den meisten Anwendungsgebieten werden Modellierungssprachen auf hoher Ebene, wie zum Beispiel SystemC, in den frühen Entwurfsphasen benutzt. Die Einführung von *Universal Verification Methodology (UVM)* half auch traditionell konservativen industriellen Feldern, auf abstraktere Hochsprachen umzusteigen. Die Automobilindustrie hat *UVM* aufgegriffen, wohingegen die Raumfahrtindustrie noch etwas zögert. Allerdings sind alle Industrien von der Entwurfskluft betroffen.

Eine virtuelle Plattform basierend auf SystemC und *Transaction-Level Modeling (TLM)* kann einen Geschwindigkeitszuwachs in der Simulation von mehreren Größenordnungen im Vergleich zu einer klassischen *Register-Transfer Level (RTL)* Simulation bewirken [5].

Der Entwurf eingebetteter Systeme mit Modellierungsmethoden auf hoher Ebene bedeutet, Hardware-Entwürfe und dazugehörige Tests in Hochsprachen wie C++, worauf SystemC basiert, zu schreiben. Der dabei entstehende Quelltext wird heutzutage üblicherweise in Versionskontrollsystemen gespeichert. Das bedeutet wiederum, dass die gesamte virtuelle Plattform von jeder Quelltextänderung betroffen sein kann und sämtliche Tests wieder überprüft werden müssen, sobald eine Änderung in das Versionskontrollsystem eingespielt wird. In der modernen Software-Entwicklung wird *Continuous Integration (CI)* benutzt um automatisiert zu überprüfen, ob eine eingespielte Änderung am Quelltext bestehende Funktionalitäten beeinträchtigt. Ein wichtiger Teil des *CI*-Konzepts ist das Prüfen von Änderungen. Entwickler und Ingenieure sind oft davon abgeneigt, ausführliche Testkampagnen zu entwickeln. Dieser Umstand kann durch die Etablierung von Testmethodiken, welche dabei helfen den Aufwand für das Schreiben vieler Tests zu reduzieren, umgangen werden.

UVM stellt eine solche Testmethodik dar.

Die Anwendung des *CI*-Konzepts auf den Entwurf und das Testen von eingebetteten Systemen fordert schnelle Bau- und Test-Ausführungszeiten von dem genutzten Framework für virtuelle Plattformen. Die Ausführungszeiten der Tests sollten geringer sein als die durchschnittliche Zeit zwischen zwei Änderungen am Quelltext. Für diesen Anwendungsfall wird auch die Fähigkeit, einen bestimmten Zustand der virtuellen Plattform zu speichern, erforderlich. Es ergibt sich zusätzlich die Erschwernis, dass der gespeicherte Zustand (engl. *checkpoint*) auch funktionieren muss, wenn sich der Quelltext des Simulationsmodells geändert hat. Weiterhin sollten die *checkpoints* portabel sein, um die Fehlersuche auf anderen Systemen oder um verteilte Simulationen zu ermöglichen. Allerdings kann das Hinzufügen einer *checkpointing* Funktionalität auch die Komplexität der Code-Basis erhöhen oder zu verlängerten Ausführungszeiten durch den *checkpointing* Prozess führen. Die zusätzliche Komplexität könnte zu längeren Compile-Zeiten führen. Die verlängerten Ausführungszeiten könnten im schlimmsten Fall den Nutzen des *checkpointing* hinfällig machen. Diese Auswirkungen auf das gesamte Framework müssen in der Entwurfsphase der *checkpointing* Funktionalität berücksichtigt und danach evaluiert werden. Das Speichern und Wiederherstellen der Zustände einer Simulation erfordert die Serialisierung aller Datenstrukturen, die sich in den Simulationsmodellen befinden. Diese serialisierten Daten müssen auf festen Speicher geschrieben werden.

Das Verbessern von Frameworks und Etablieren besserer Methodiken hilft nur die Entwurfs-Kluft zu verringern, wenn diese Änderungen mit Berücksichtigung der Bedürfnisse der Entwickler und Ingenieure eingeführt werden. Letztendlich ist es ihre Produktivität, die gesteigert werden soll. Entwickler und Ingenieure müssen Zeit aufwenden, um neue Methodiken und Werkzeuge zu erlernen. Diese Zeit kann reduziert werden, indem nicht nur auf technische Funktionalitäten geachtet wird, sondern auch auf die Benutzbarkeit ebendieser Funktionalitäten. Wenn Modellentwickler sehr viel zusätzlichen Code schreiben müssen, um neue Funktionalitäten, wie zum Beispiel *checkpointing*, zu ermöglichen, wird ihre Produktivität nicht gesteigert. Daher sollten neue Funktionalitäten auf eine Art implementiert werden, die für den Benutzer transparent ist. Zur gleichen Zeit möchten erfahrene Benutzer die Implementierung der Funktionalitäten an ihre Bedürfnisse anpassen. Diese Möglichkeit sollte ihnen während der Entwurfsphase nicht verbaut werden.

Die Fähigkeit den Zustand einer virtuellen Plattform zu speichern, ermöglicht es den Entwicklern, längere Testkampagnen laufen zu lassen, die auch zufällig erzeugte Teststimuli beinhalten können oder, falls die gespeicherten Zustände modifizierbar sind, fehlerbehaftete Zustände in die Simulationsmodelle zu injizieren.

Das SoCRocket Framework für virtuelle Plattformen beinhaltet bereits alle nötigen Bausteine, um komplexe eingebettete Systeme zu simulieren und Treiber oder andere Software-Anwendungen auf den simulierten Systemen zu entwickeln. Wie bei jedem Simulationsframework, welches nicht auf der reinen funktionalen Ebene operiert, dauert das Starten eines Betriebssystems eine gewisse Zeit. Diese Zeit macht es schwierig, komplexe Teststrategien effizient auszuführen. Weiterhin werden Software-Entwickler in ihrer Produktivität behindert, wenn sie ständig auf einen bestimmten Simulationszustand warten müssen.

Mein mit dieser Arbeit geleisteter Beitrag beinhaltet die Erweiterung des SoCRocket Frameworks um *checkpointing* Funktionalität im Sinne einer Referenzimplementierung. Diese Referenzimplementierung hält sich an die SystemC/TLM Standards und wird daher

mit anderen Implementierungen des SystemC Kernels kompatibel bleiben. Zusätzlich wird die Referenz-SystemC-Implementierung der *UVM* Bibliothek in das SoCRocket Framework integriert, um zu zeigen, wie Tests effizienter gemacht werden können, wenn die virtuelle Plattform *checkpointing* unterstützt. Die Kombination von SystemC *checkpointing* und *UVM* Tests ist der beste Weg, um virtuelle Plattformen in eine *CI*-Umgebung zu integrieren. Diese Beiträge werden helfen, die Entwurfs-Kluft zu verringern, indem Ingenieure, Tester und Entwickler befähigt werden sehr viel effizienter zu arbeiten. So werden moderne Methoden aus dem Entwurf eingebetteter Systeme mit aktuellen Methoden aus der Software-Entwicklung zusammengebracht.

Das *snapshotting* Framework, welches in dieser Arbeit entwickelt und präsentiert wird, baut auf die bestehenden SystemC Standards auf. Modellentwickler können zu den Standards kompatible Vorlagen und Code-Generatoren mit nur leichten Modifikationen benutzen, um neue Modelle mit eingebauter *snapshotting* Funktionalität zu erzeugen.

Die Evaluation hat die hohe Wiederverwendbarkeit des hier präsentierten *UVM Testbench* Codes bewiesen. Die *UVM* SystemC Bibliothek, die in dieser Arbeit benutzt wurde, ist ein frühes Release, welches mit weiteren Iterationen sicherlich weiterhin optimiert wird. Diese Verbesserungen werden zu effizienterem *Testbench* Code führen, der dann vollständig generisch implementiert werden kann. Solch generischer *Testbench* Code mit eingebauter *snapshotting* Funktionalität wird die Test-Ausführungszeiten erheblich verbessern sowie die Testentwicklung an sich deutlich erleichtern.

Meine *snapshotting* Herangehensweise ist der Standardisierungs-Roadmap der *Configuration, Control & Inspection (CCI)* Arbeitsgruppe innerhalb Accellera voraus. Die Standardisierung des Speicherns und Wiederherstellens von Zuständen in virtuellen Plattformen ist noch weit entfernt, wenn man als Maßstab die Standardisierung der Konfigurationsparameter zu Grunde legt. Das SoCRocket Framework, welches ich in meiner Implementierung benutze, verwendet schon einen Vorreiter der standardisierten Konfigurationsparameter. Weiterhin basiert die Registerimplementierung, die ich für die Serialisierung benutzt habe, auf einem Vorschlag von Cadence für eine standardisierte Implementierung. Da Cadence starken Einfluss auf die Standardisierung genießt, ist davon auszugehen, dass die finale standardisierte Registerimplementierung sehr nah an der aktuell in SoCRocket genutzten Implementierung ist.

Diese Arbeit steuert dem SoCRocket Framework für virtuelle Plattformen eine Erweiterung bei, die es ermöglicht, den Zustand der Simulation zu speichern und wiederherzustellen. Diese Erweiterung kann als Referenzimplementierung einer solchen Funktionalität angesehen werden, da ausschließlich SystemC/TLM Standards benutzt wurden. Somit ist die Implementierung kompatibel zu anderen Frameworks. Da die hier präsentierte Implementierung bereits standardisierte Schnittstellen für Konfigurationsparameter und Register benutzt, ist sie der perfekte Kandidat für die Standardisierung der Speichern- und Wiederherstellen-Funktionalität für virtuelle Plattformen. Weiterhin ermöglicht die Integration der *UVM* SystemC Bibliothek in das Framework die Umsetzung der testgetriebenen Entwicklung und schnelle Validierung von SystemC/TLM Modellen mit Hilfe von *snapshots*. Diese Erweiterungen verringern die Entwurfs-Kluft und ermöglichen es Designern, Entwicklern und Testern effizienter zu arbeiten.

Acknowledgements

First of all, I would like to thank Prof. Dr.-Ing. Mladen Berekovic for advising me on this thesis and for his continuous support of the research on the SoCRocket SystemC framework.

I would also like to thank Dr.-Ing. Rainer Buchty for his valuable feedback on my research and especially during the writing process of the thesis.

Further thanks are in order for my coworkers Rolf Meyer, Jan Wagner and Sven Horsinka for our many engaging discussions related to our theses topics and other unrelated technical interests as well as for their helpful feedback.

Finally, I would like to thank my family, my friends and especially my wife for all the support they have given me.

Contents

Abbreviations

ADL argument-dependent lookup. 61, 77

AHB Advanced High-performance Bus. 16, 113

AMBA Advanced Microcontroller Bus Architecture. 113

AMS Analog Mixed Signal. 45

APB Advanced Peripheral Bus. 16, 113

API Application Programming Interface. 16, 17, 41, 42, 44–46, 50, 51, 65, 110, 121, 126

CCI Configuration, Control & Inspection. ix, 17, 45, 46, 50, 51, 54, 55, 67, 122–125, 132

CI Continuous Integration. 2–4, 6, 7, 9, 17–20, 41, 42, 46, 52, 53, 55, 89, 126, 127, 129–131

DMTCP Distributed Multi-Threaded Checkpointing. 34, 89, 103, 104, 107–113, 121

DUT Device under Test. xi, xii, 22, 23, 44, 45, 89–97, 99–101, 109, 113, 115, 118, 122, 131, 142

EDA Electronic Design Automation. 3, 5, 43, 46, 54

ESA European Space Agency. 1, 3, 5

ESLD Electronic System-Level Design. ix, 7, 11–13, 20, 37

FPGA Field Programmable Gate Array. 45, 127

GPL GNU General Public License. 113

HDL Hardware-Description Language. 1, 4, 11, 12, 14, 15, 127

HPC high-perfomance computing. 10

IC Integrated Circuit. 12

IP Intellectual Property. 44

IRQ interrupt request. 89, 113, 115

IRQMP Multiprocessor Interrupt Controller. ii, ix, xi, xii, 89, 113–118, 122, 142

JSON JavaScript Object Notification. 11, 40, 41, 50, 51, 82, 84, 101, 113, 121, 125, 131

MTCP Multi-Threaded Checkpointing. 34, 107

OS operating system. 1, 2, 4, 10, 29, 32, 43, 47, 48, 50, 53, 55, 58

RTL Register-Transfer Level. 1, 6, 11–13, 15, 35, 41, 50, 52, 72, 122, 124, 126

SoC System on Chip. 3, 6, 14, 44, 45, 113, 114

TLC Thread-based Live Checkpointing. 30, 31

TLM Transaction-Level Modeling. 1, 6, 12, 13, 16, 20, 23, 28, 40, 44–46, 72, 73, 89, 90, 94, 124–126

USI Universal Scripting Interface. ix, 6, 15–17, 47, 55, 57, 58, 60, 65–67, 87, 103–105, 108, 109, 122, 125–127

UVM Universal Verification Methodology. ix, xi, xii, 1, 2, 4, 5, 7, 9, 20–24, 41, 44–46, 52, 89–94, 96–105, 108, 109, 113, 115–118, 122–124, 127, 129, 131–136, 142

VM Virtual Machine. 29–32

XML Extensible Markup Language. 84

List of Tables

List of Figures

List of Listings

1 Introduction

The steady increase in complexity of high-end embedded systems goes along with an increasingly complex design process. Electronic system-design started with simple schematic design. When schematics became too complex hardware description languages were introduced to transform textual descriptions of functionality into schematic designs. After a while designing complete systems in hardware description languages reached a level of complexity where they became virtually unusable. The next level of abstraction was introduced, SystemC and Transaction Level Modelling [1][2]. We are currently still in this transition phase from Hardware-Description Language (HDL) based design towards virtual-platform-based design of embedded systems. Development complexity has seen a threefold growth between the years 2003 and 2013 [3]. As design complexity rises faster than developer productivity a gap forms. This dissonance has been described in 2007 by the International Technology Roadmap for Semiconductors as the design gap [4].

In the last five years, this growth in complexity has not been slowed down and the design gap has widened. To alleviate the negative effects towards productivity, the apparent solutions would be to hire more personnel, extend duration of projects, and reuse more IP cores and designs. However, large teams diminish overall productivity dramatically [3]. Restoring productivity while at the same time managing increased design complexity can also be achieved through focussing on the development of new tools and design methodologies.

In most application areas, high-level modelling languages such as SystemC are used in early design phases. In the consumer electronics field which is usually fast-paced, high-level modelling is used to shorten production cycles. In aerospace or automotive electronics, which are traditionally slow-paced and resisting change, the adoption has been hesitant. The slower production cycles stem from rigorous testing and verification which are required to meet certification standards. These areas could benefit tremendously from using high-level methodologies that support testing and verification.

The introduction of Universal Verification Methodology (UVM) helped rather conservative industry fields to switch to higher-level language. The automotive industry has embraced UVM while the aerospace industry is still not ready. However, the design gap affects all industries. The aerospace field is especially burdened with regulations and requirements to enforce reliability in designs. Complying with these requirements is only possible through comprehensive testing and validation. A virtual platform framework like SoCRocket, which has been developed in conjunction with the European Space Agency (ESA), is a first step towards adopting high-level modelling methodologies.

A virtual platform based on SystemC and Transaction-Level Modeling (TLM) can achieve a speed-up in simulation time of several orders of magnitude compared to a Register-Transfer Level (RTL) simulation [5]. Practically, this means that an operating system (OS) can be booted within the virtual platform in one hour compared to a whole day for the same task in RTL. Of course increasing the abstraction level and simulation speed comes with the price of lost accuracy. For most use cases like design space exploration or early testing of driver software, this accuracy is sufficient.

There are use cases where booting an OS in an hour is still not fast enough. Moreover, I have mentioned above the need for comprehensive testing in the aerospace domain. Imagine a test session that involves booting up an OS, starting a test application and checking the results. Multiply this by a thousand possible configuration variants for the simulated system and the one hour time to finish the simulation looks quite dismal. There might also be the requirement to test more accurate model implementations which run generally much slower, but can be used for verification purposes. These kinds of models are used during testing of driver code, real-time OS or other low-level functionality where timing needs to be accurate. One solution to this kind of use case would be acquiring more compute resources and run the simulations in parallel. Another solution would be to reduce the simulation time through checkpointing the virtual platform state where it diverges into the multiple variants and use the saved state to start subsequent simulations. The former is quite expensive, the latter is not available yet.

Designing embedded systems using high-level modelling methodologies means writing hardware design and accompanying tests in high-level programming languages such as C++, which SystemC is based on. This code is nowadays usually stored in version control systems. This in turn means that with each change in the code base, the whole virtual platform for the embedded system might be affected and all test cases need to be rechecked. In modern software development Continuous Integration (CI) is used to automatically test if a submitted piece of code breaks functionality. An important part of the CI concept is testing. Developers and engineers are often reluctant to write extensive tests. This can be circumvented by establishing testing methodologies like UVM which help reduce the effort in writing a large number of tests.

Application of the CI concept to embedded system design and testing requires fast build and test execution times from the virtual platform framework. Fast build times are achieved through using shared build caches. Test execution times should be lower than the average time between two code changes. Also for this use case the ability to save a specific state of a virtual platform becomes necessary. With the added complication that a checkpoint needs to work even if the simulation-model-code changes. Furthermore, checkpoints should be *portable* in order to debug issues or simply run distributed simulations from the same base checkpoint. However, adding checkpointing functionality might add complexity to the code base or cause overhead through the checkpointing process during execution. Additional code complexity could lead to longer build times. Whereas the added overhead could, in the worst case, outweigh the benefit of introducing checkpointing. These impacts on overall framework *performance* have to be considered during the design phase and will be evaluated afterwards.

Nowadays complexity in software frameworks is often introduced through dependency bloat. Dependency bloat can best be observed in app and web development frameworks. Modern programming languages such as JavaScript or Python come with powerful package managers and vast repositories of libraries for developers to choose from. The problem with this is, that there are many libraries that solve mundane programming problems while at the same time introducing a long dependency tree to the project. Maintaining projects with such complex dependencies is no easy feat. Second or third level dependencies are often overlooked and could introduce unwanted bugs or even security issues in the code base.

Fortunately, this trend as not yet reached C++. Creating *self-reliant* virtual platform frameworks with a minimal number of dependencies is still possible. Especially header-only extensions that can easily be included and distributed with the platform code help maintaining a small dependency tree and containing complexity. Thereby keeping the platform compatible and portable.

Improving frameworks and establishing better methodologies helps reducing the design gap only when the changes are introduced with developers and engineers in mind. After all, it is their productivity that wants improvement. Developers and engineers have to spend time learning new tools and methodologies. This time can be reduced by focussing not only on technical features, but also on the usability of those features. If model developers need to write a lot of extra code to enable new features, e.g. checkpointing, their productivity will not improve. Hence, new features have to be implemented in a way that is *transparent* to the user. At the same time advanced user might want to customize the implementation of new features to their exact needs. This path should not be blocked during the design phase.

Usability does not only consist of ease of use. Electronic Design Automation (EDA) software tends to be very complex, which can lead to frustrations for the user due to unpredictable behaviour of the application or user errors due to design complexity of the application itself. Therefore, applications have to be designed in a way that allows for *reliable* operation. If something works today, it should also work tomorrow in the same way. Keeping the frustration level low can further improve productivity.

Reliable operation can be ensured through extensive testing of the virtual platform framework itself as well as its models inside a CI workflow.

Unit testing for simulation models is a fast way of ensuring adherence to the specification. Although, unit tests only cover, as the name suggests, very small units of the virtual platform, e.g. a single model. Comprehensive test campaigns will also contain tests where multiple simulation models interact or in the case of operating-system-level tests, the whole system needs to be simulated. The whole system while change behaviour depending on the enabled components and their configurations.

Having the ability to checkpoint the state of a virtual platform enables engineers to run larger test campaigns, that can also involve randomized test stimuli or, if the checkpoints are *modifiable*, injection of faulty states into the simulation models.

The entry point into high-level modelling with SystemC/TLM is the reference implementation available as open source from Accellera [6]. However, the reference implementation only comes with the bare essentials. A virtual platform framework, such as SoCRocket, offers a better starting point for high-level modelling. The virtual platform comes already with the models for a complete System on Chip (SoC) based on Gaisler's GRLib and LEON3 CPU [7]. Almost all of the models included with SoCRocket as SystemC/TLM have an equivalent in Gaisler's GRLib as VHDL implementation. During the early development phase together with ESA, the modelling methodology was developed in conjunction with the platform and supporting tools. The extensive model library helps in testing and verification as there is always a reference implementation available to test against. When the initial version was released the SoCRocket virtual platform framework offered very efficient modelling techniques to create new IP models with. Later the platform was enhanced with extensive debugging capabilities through an advanced logging and tracing framework. Along with this logging and tracing framework, a powerful scripting interface was introduced, which

enabled control of the running simulation from various scripting languages. This allowed for more efficient handling of debug and logging information. There is ongoing work to enhance the scripting interface with the ability to inject faults directly into simulation models. The scripting interface also exposes some internal model parameters, which makes it possible to instrument models with reference power values and subsequently run power-aware design space exploration. Another way of using the scripting interface and the exposed model configuration parameters is to feed a HDL code generator to generate HDL models from explored platform configurations. The existence of VHDL implementations for the GRLib models makes this possible. This technique in turn can be used to gather reference values for power or area for the models.

The SoCRocket virtual platform framework comes already with all pieces to simulate complex embedded systems and to develop drivers or software applications on top. As with any other simulation framework that does not work at the purely functional level, booting up an OS takes some time. This time makes it difficult to run complex test strategies efficiently. Furthermore, software developers are slowed down, when they always have to wait for the simulation to reach a desired state.

In this work, I will present a way towards making SystemC/TLM simulation more efficient through addition of checkpointing functionality. Furthermore, I will show how adding checkpointing to a virtual platform framework opens the possibility of running automated tests in a CI environment. The tests will be implemented using the UVM library for SystemC.

My contributions with this work include extending the SoCRocket virtual platform framework with checkpointing functionality as a reference implementation. This reference implementation will adhere to SystemC/TLM standards and thereby stay compatible with other implementations of the SystemC Kernel. Additionally, I will integrate the reference UVM SystemC implementation into SoCRocket to show how tests can be made more efficient when the virtual platform supports checkpointing. The combination of SystemC checkpointing and UVM testing is the best way to integrate virtual platforms into a CI environment. These contributions will help narrowing the design gap a bit by enabling designers, testers and developers to work much more efficiently.

This thesis is structured as follows: In the next Chapter, I will give more details about the motivation of this work. In the subsequent Chapter, I will explain the fundamentals needed to understand the remainder of the Chapters. Furthermore, I will give an overview of checkpointing, serialization, virtual platforms, continuous integration as well as the universal verification methodology. In Chapter 4, I present a small history of checkpointing mechanisms and their implementations. Followed by necessary improvements in C++ to enable development of checkpointing functionality. An overview of current serialization options will be presented. Furthermore, I will look into what constitutes continuous integration for virtual platforms. Finally, I will show the current state of SystemC checkpointing. The architecture of my checkpointing solution is described in Chapter 5. Chapter 6 contains details about the snapshotting implementation in SoCRocket as well as the extensions needed for including the UVM library. The implementation Chapter is followed by the evaluation. In the evaluation, my solution will be compared to a state of the art user-level checkpointing implementation. Furthermore, my own implementation is tested with a real world use case of running tests against a specific model. The penultimate Chapter discusses of the evaluation results. Finally, this work is summarized in Chapter 9.

2 Motivation

In the previous Section, I have already described the design gap and its implications for engineers and EDA tools. The International Technology Roadmap for Semiconductors [4] considers tools helping developers debug and understand systems a grand industry challenge. The software aspect of embedded-system development currently can make up about half of the total design costs.

Hardware/Software Co-Design needs the ability to focus both on hardware and software. Therefore, software development and hardware testing is often done with the help of virtual platforms. To speed up testing of software and debugging, a mechanism to save the state of a running simulation is needed. During testing, it should be possible to inject desired internal states directly into the models and thereby reduce execution time for single tests.

Simulation is important during testing for finding errors. Simulation needs to be able to cover all cases. One option used mainly in security research to break software is fuzzing, where a pseudo-random sequence of values is tested. Another approach is supplying faulty states of simulated systems as checkpoints. The Shiaparelli failure, where the ESA has lost a Mars lander due to unforeseen technical behaviour [8], has shown importance of wide range testing. The scientists did not test all possible environmental parameters for the sensors. Lack of supporting tools may have been one reason for this. Especially the software could have been exercised in the right simulation environment to test also unexpected environmental conditions with fuzzed or randomized sensor data.

With the proliferation of testing methodologies like UVM or test-driven design, the focus on testing during development cycles is getting larger and larger. Testing has always been important during hardware development, and UVM has enabled engineers to write test frameworks and tests efficiently. Having the support from big EDA vendors, the methodology found its way into several tools and is now widely used. In the year 2017 it was accepted as an IEEE standard.

In software development the occurrence and availability of compute clusters, e.g. cloud, has helped bringing along a trend towards continuous testing with the aid of automated tools. Now, this trend is finding its way into hardware development as well, although hardware simulations are not quite suited for daily runs, since they might take more than a day to complete. Here, SystemC with its more abstract modelling and good-enough accuracy can shine, since it is mostly used for early exploration and driver/software development and is well suited for the integration into continuous integration frameworks. The software developers know these tools already and can develop software in their familiar environment, while the hardware developers can stay in their familiar environment as well.

Booting up operating systems can still take too much time, even with loosely-timed SystemC simulations. Hence, there is a demand for storing a specific simulation state, like a booted operating system, and continuing from that state with application testing. Having a distributed simulation framework with checkpointing support would also enable checkpoints to be created on more powerful dedicated machines to be subsequently transferred to developer machines for detailed inspection or further simulation.

Current SystemC checkpointing mechanisms save the entire process state or rely on commercial, functional simulators and heavily modified models. They are only suitable for resuming work and maybe complex debugging tasks, but for working with internal states of modules and maybe even changing them before restoring, developers need more flexibility and insights into the models.

Improving on the current state requires abandoning the idea of using functional simulators to introduce checkpointing into SystemC. Modifying existing models is also undesirable and can be avoided. My approach for snapshotting SystemC simulations follows the application-level checkpointing approach. Application level means that the SystemC virtual platform itself will be able to handle the checkpointing process, without the need for external tools. This could mean that the SystemC simulation kernel needs to be modified, but I will not follow that path and instead merely augment the SystemC kernel with the checkpointing functionality. The goal is to implement the checkpointing framework in a way that is transparent to the user of the virtual platform.

In the previous Chapter, I have already mentioned the advantages of using CI during embedded system development. The CI concept works best, when build times and test execution times are very fast. To fulfill the requirement of fast build times, I will keep the complexity of the checkpointing framework simple. Test execution times are already improved through the introduction of checkpointing.

Using checkpoints for debugging is quite common with commercial simulators as can be seen in Chapter 4, but using it during validation and testing with a unified framework and distributed processing approach has not been done yet. So far comprehensive analysis capabilities have been missing in commercial simulators and virtual platforms. They allow inspection of the platform models during runtime and can be used to implement checkpointing functionality. The virtual platform framework SoCRocket, which I use in this thesis already gained good analytical features in previous works upon which a checkpointing framework can be built. Currently these introspection and reflection extensions in SoCRocket are used for the Universal Scripting Interface.

The introspection and reflection extensions in SoCRocket help analyse models using test benches or scripts. They are needed to inspect internal model states during any point in the simulation, save the states and where applicable restore the states. Saving and restoring these states requires serializing the data structures contained within and writing them to storage. As we learn in the next Chapter, serialization is no trivial problem. It is neither a new problem, so there are several options available for implementation of serialization.

Depending on the design-abstraction-level used for modelling the designers encounter different obstacles. The biggest obstacle at the RTL right now is the sheer complexity of modern System on Chip designs. This complexity can be reigned in by moving to a higher abstraction level such as TLM. With RTL it was still fairly easy to checkpoint simulations as the state was clearly defined in the components and could be automatically discovered. With the higher abstraction level of TLM and the usage of modern high-level programming languages the complexity of the components is hidden away from the engineer. Which makes it challenging to implement checkpointing that works transparently and without putting too much burden on the model developer.

The trend for test driven development has been mentioned before. This trend lends itself very well towards development of SystemC/TLM models. Test driven development uses

CI tools to drive tests. In Electronic System-Level Design (ESLD) a CI can be used for fast verification of models that are still being developed. When the specification is available, test engineers can already write tests using the UVM SystemC library. Using the library they can reuse lots of test bench components and can concentrate on writing actual tests. So when the specification changes or model code is being refactored, the test code can follow without much effort. The structure of UVM test benches helps with reusing all kinds of typical test bench components. Even sequences of test stimuli can be reused. In this scenario, checkpointing helps in setting up the simulated models and getting them directly into the desired state for running tests. With portable checkpoints it is furthermore possible to distribute the workload and run many tests in parallel. The ability to modify checkpoint data will enable fuzzing tests with randomized stimuli. In the evaluation of my checkpointing solution, I will revisit this scenario.

In the next Chapter, I will give an overview of the fundamental technologies that are needed to implement an efficient checkpointing framework for SystemC/TLM virtual platforms.

3 Fundamentals

In this Chapter, I will explain the fundamentals necessary for understanding the remainder of the thesis. Good understanding of the programming language C++ and Python is assumed. A short overview of checkpointing and serialization will be given, which will be further elaborated in the next Chapter. Afterwards a description of modern hardware design and verification will follow, starting with a brief description of SystemC/TLM and the used SoCRocket virtual platform framework is also considered necessary. The use-case example following later in the thesis relies heavily on the Universal Verification Methodology standard and the Continuous Integration concept. Since these topics are quite complex, their description is included in this Chapter.

3.1 Checkpointing

In this Section, I will give an overview about common *checkpointing* methods in their historic context. Before going into any detail about *checkpointing* though, we have to clarify the distinction (if any) between a *snapshot* and a *checkpoint*.

The dictionary [9] defines the terms as follows:

checkpoint
: A barrier or manned entrance, typically at a border, where security checks are carried out on travellers.
  - A place on the route in a long-distance race where the time for each competitor is recorded.
  - A location whose exact position can be verified visually or electronically, used by pilots to aid navigation.

snapshot
: An informal photograph taken quickly, typically with a small handheld camera.
  - A brief look or summary.
  - (*Computing*) a record of the contents of a storage location or data file at a given time.

Looking at the execution of an application as a long-distance race, we can deduct, that checkpointing would be the act of recording the current state of the participants. In this analogy, the participants of our application would be the objects stored in its memory. Since a *snapshot* is defined as "record of the contents of a storage location", our act of checkpointing would produce a snapshot of the application's memory. Hence, *checkpointing* and *snapshotting* could be used synonymously here, as it also has been in many scientific works.

Now that the terminology is clear, we can have a look at where and how checkpointing can be applied. First we have to look at the different abstraction levels. In the case of checkpointing, the user level is considered the highest abstraction, and system level the lowest.

Different levels where checkpointing can be applied:

User
Link code to checkpointing library containing the necessary functionality, *libckpt* is a good example for this [10]. A multi-threaded variant is presented in [11].

Application
Code is inserted directly to the application to save immediate results to checkpoint files. Examples of application-level checkpointing can be found in [12] and [13].

Virtual Machine
The hypervisor is modified to enable checkpointing of virtual machines. KVM [14] has simple checkpointing built-in, using the stop-and-copy mechanism described below. For the Xen hypervisor, multiple solutions exist, for example REMUS [15].

System
Checkpointing, integrated at OS level, e.g. kernel or file system support. Kernel-level examples are BLCR [16] or TICK [17]. File systems with integrated checkpointing support are for example ZFS [18] or BTRFS [19].

Each level has its own benefits and drawbacks. Application-level checkpointing can benefit from compilation and static code-analysis tricks, but it lacks user transparency. The user-level provides more transparency, although using an external library offers less flexibility. System-level provides some amount of transparency, but is highly dependent on the used OS. Using checkpointing on virtual-machine level offers the most transparency with the drawback of having to package applications in virtual machines with OS overhead.

Apart from different abstraction levels, where checkpointing can be applied, several common checkpointing mechanisms have been established:

stop-and-copy
One of the most common checkpointing mechanisms. Execution of the process is halted while the entire state is saved to a checkpoint file. As mentioned before, this approach is implemented in the KVM hypervisor.

incremental checkpointing
Slightly optimized mechanism. Only changes since the last checkpoint are saved. Still needs to halt the process. The previously mentioned TICK [17] kernel extension uses this mechanism.

diskless checkpointing
Can be used as a further optimization of incremental checkpointing. Instead of writing to disk checkpoint information is only stored in memory. This was first described by Plank et. al in [20].

multi-level checkpointing
A combination of disk-less and stop-and-copy checkpointing. The state is saved to a RAM disk in short intervals; periodically. The RAM disk is flushed to real disk storage. This approach is commonly used for high-perfomance computing (HPC) systems. Moody et. al describe their implementation of a multi-level checkpointing library for the Lawrence Livermore National Laboratory in [21].

concurrent checkpointing
: Multi-threaded mechanism that launches new threads for copy and write operations. Several possible implementations of this mechanism are explored in [22].

copy-on-write
: This mechanism uses a cloned process created by the *fork* system call to write memory contents to disk while the original process continues running. One of the first implementations of this mechanism was in the aforementioned *libckpt* library described in [10].

Using the term *snapshotting* over *checkpointing* seems fitting, since I describe a methodology for saving the current state of a simulation. A *checkpointing* mechanism can use a *snapshotting* mechanism to get the data for its *checkpoints*.

Most of these checkpointing mechanisms make use of serialization which will be explained in the next Section.

3.2 Serialization

Saving the state of a complex system such as a SystemC simulation requires consolidating the various data spread among memory and likely reorder it. This process is usually referred to as serialization. Serializing primitive data types like *int* or *float* is trivial, the data can be written as it is stored in memory. The difficulty is serializing pointers; here, the serializer needs knowledge of what the pointer points to and how to serialize and reconstruct the data. With this complexity, it does not make sense to write own custom code to serialize objects.

As to the format for storing the serialized data in, several data formats have been standardized. Well known representatives are XML and JavaScript Object Notification (JSON). The latter one is described in Section 6.2.4.

Several modern programming languages have support for serialization built-in by design, for example Python or Java. Unfortunately, C++ is not included in that group. Since serialization is a very useful and often required feature, a plethora of libraries exists that add serialization support to C++. The difficulty lies in deciding on one library that best suits the current project.

3.3 Virtual Platforms

ESLD is an electronic design-methodology that is vaguely one step above RTL or rather one abstraction level higher than RTL design.

Gajski and Kuhn have developed a way of depicting the different abstraction levels of hardware design in 1983. They depict the different levels as concentric rings. Three lines representing the different domains protrude from the center of the concentric rings and thereby form a y-shape. To better illustrate modern hardware design methodology, I have slightly adapted their original diagram. Figure 3.1 displays my interpretation of the classic *Gajski-Kuhn-Chart*. The three domains are in our adaptation communication, behaviour, and time. The intersections where abstraction-level rings and domain lines intersect represent the elements used in that particular domain to realize an abstraction level.

ESLD uses high-level languages such as C, C++ or SystemC instead of HDLs like VHDL or Verilog. Another way to think about ESLD is that it depicts the simultaneous design of

hardware and software. Therefore, ESLD can be placed between system level and transaction level in Figure 3.1.

This Section will give a brief overview over the history of design strategies for electronic systems, including RTL and TLM, the latter being a form of ESLD.

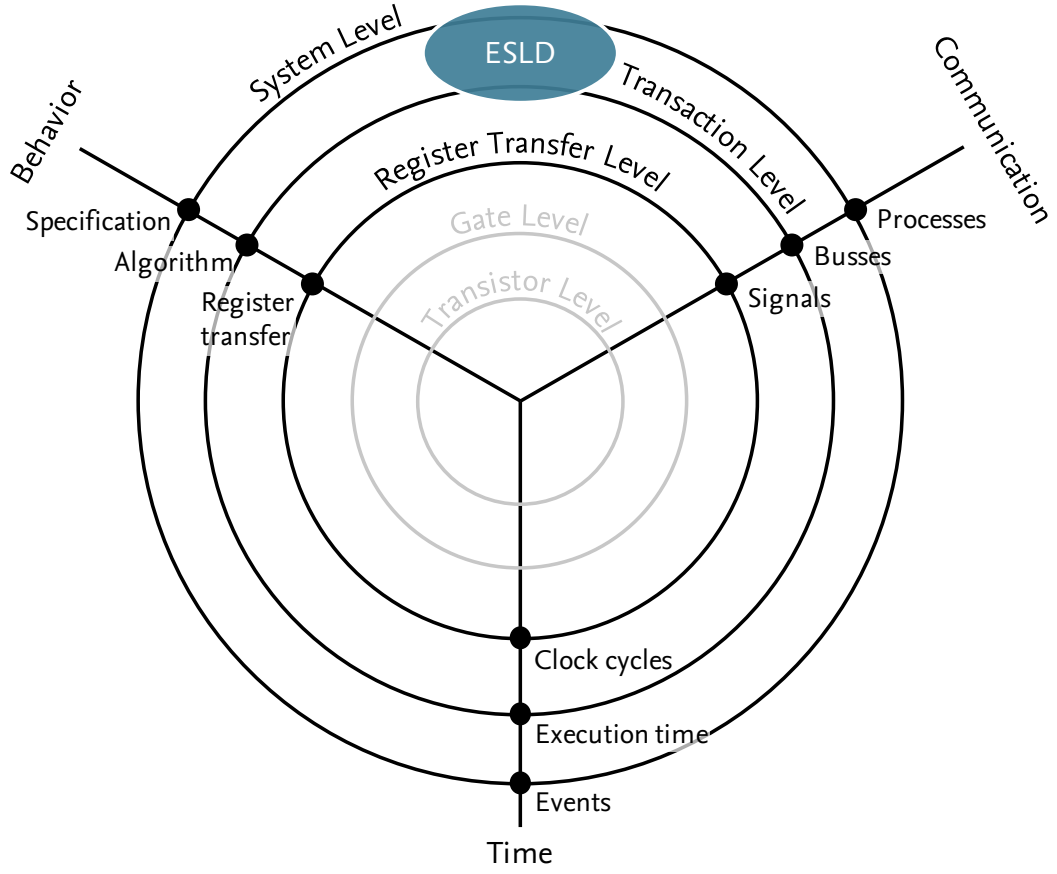

Figure 3.1: Modified Gajski-Kuhn chart depicting design abstraction-levels and the classification of ESLD

Mostly schematics were used in early days of Integrated Circuit (IC) design. As the ICs grew more complex over the time, standardized design approaches were searched. The design of electronic systems shifted towards the use of HDLs such as VHDL and Verilog, which operate at RTL.

The evolution of ICs has not come to a halt, ICs grow more complex every year, and slowly the usage of equally complex HDLs has become a bottleneck: Simulations for complex systems in an HDL take a lot of time. Hardware and software development are strictly separated. Therefore, the high-level approach TLM was developed. TLM combines hardware and software development, as SystemC is often used for this approach. Moreover, TLM improves simulation times drastically through abstraction of the hardware models.

The most widely used abstraction levels during development are RTL and TLM:

RTL

The RTL is a design-abstraction-level nearer to hardware than ESLD. Usually, the sequential logic of electronic hardware is designed in form of registers, i.e. D-flip-flops. Register state transitions are defined by combinatorial logic operations. RTL as a whole abstracts structure, logic and timing.

With logic synthesis tools such as the Synopsys Design Compiler, an RTL description of a System written in e.g. VHDL can be converted to a gate-level description. The result is a netlist, which can be used in the place-and-route process to create a physical layout.

TLM

TLM is a high-level approach to describe the functionality of a hardware system and can be counted towards ESLD. TLM is about the abstraction of communication and realised with the help of SystemC. Figure 3.1 shows that transaction-level modeling is achieved through focusing on modeling *buses* using *algorithms* and only keeping track of *execution time*. The latest IEEE standard revision is SystemC/TLM 2.0 which was released in 2012 [23].

The TLM 2.0 standard includes various levels of accuracy. Transactions may be modelled approximately-timed, keeping each model synchronized to a common clock. This is a non-blocking way of communication, as each model is synchronized. Another way to model the communications is the loosely-timed approach, in which each model runs at their own time. Each thread inside a model will need to keep track of its time when using *temporal decoupling*. When two models need to exchange data, while being connected to the same bus for example, they need to synchronize their local times within the threads handling the bus communication.

With SystemC and TLM, hardware and software co-development can be realised. RTL-like behaviour can also be implemented with SystemC and TLM, making TLM very flexible in terms of abstraction.

Especially during debugging and testing of software, the benefits of SystemC and TLM become apparent. Through the flexible abstraction levels, the simulation speed can be very high and the developer can very efficiently get debugging information from the simulated system as would be possible with debug probes at RTL.

There are commercial implementations of the SystemC/TLM simulator from the big vendors like Synopsys and Cadence as well as open-source implementations. The Accellera reference simulator is mostly used by open-source solutions and frameworks. One such framework, which was developed mainly at TU Braunschweig, will be described in the next Section.

3.3.1 SoCRocket

The SoCRocket virtual platform framework used and extended in this thesis is completely standard-compliant to the latest SystemC/TLM standard. It was developed at TU Braunschweig over the course of several years.

SoCRocket was initially developed in conjunction with the European Space Agency (ESA) and is therefore focused on simulating SoCs used in aerospace systems. Specifically it was designed to simulate the *LEON3* processor and several peripherals included in Gaisler's *GRLIB* [7]. Like with many virtual platforms, the goal was to enhance efficiency in the design process of *LEON3*-based embedded systems. It was selected as the official virtual platform for ESA [24]. During the EU-funded EMC2 project [25], the platform was made available under the AGPL licence and is now hosted on Github [26] where it is actively maintained by the core developers.

Unlike most open-source virtual platforms, SoCRocket offers both open-source HDL and SystemC models for all its hardware components. This gives developers and researchers unique opportunities to gain insights into the inner workings of embedded systems. The SoCRocket framework is suitable for design-space exploration along with early software development and driver testing. Furthermore, automated hardware/software co-design is supported. Thanks to the openness of the platform and its models, performance analysis is possible for virtually any metric (given that reference data exists).

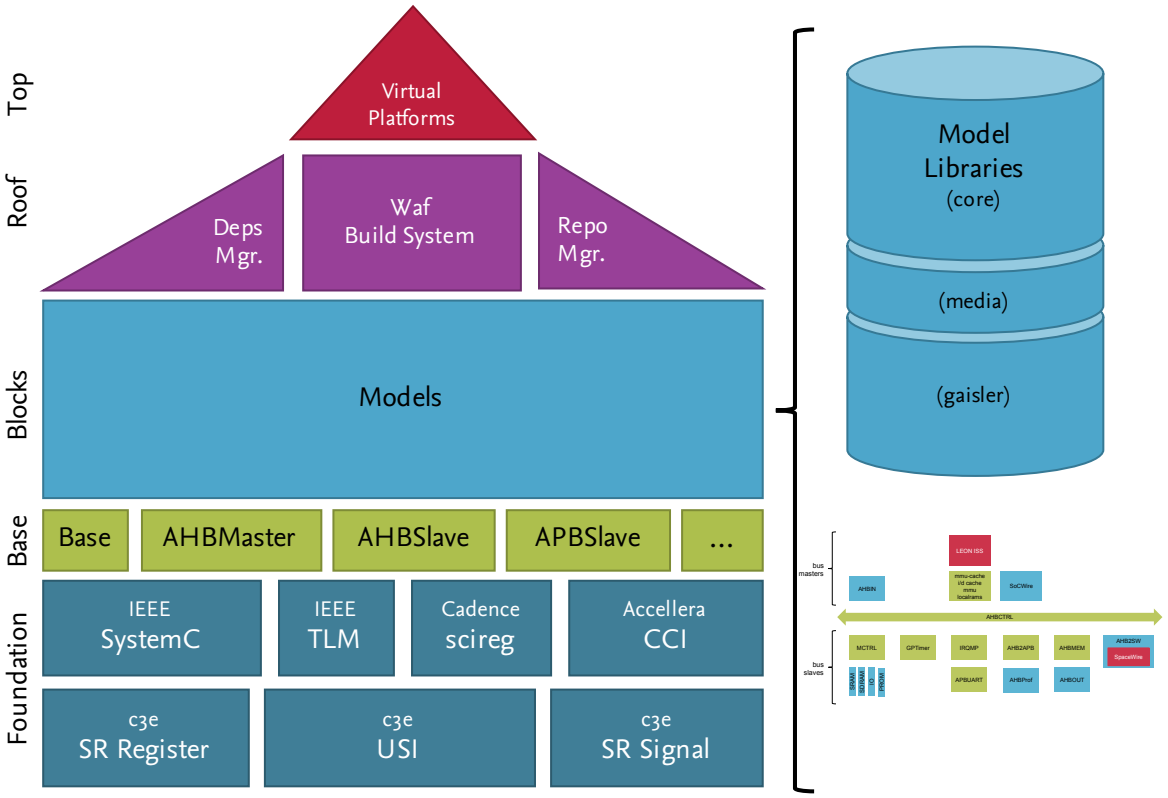

Figure 3.2: Overview of the SoCRocket building blocks adapted from [27, 28]

The basic building blocks and other components making up the SoCRocket framework are shown in Figure 3.2. The framework is built on a strong foundation of IEEE standards and proposals from industry partners like Cadence or consortiums like Accellera. For nonstandard features, the foundation is supported by solutions developed in-house. These components are described in later Sections in more detail. Sitting on top of the foundation are the base components. "Base" constitutes a bare bones model from which other models can be derived. The AHB and APB components make up the modeled bus system which is

available in loosely-timed as well as approximately-timed modeling styles. These models serve as the base for user-specific models. As mentioned before, many components from Gaisler's GRLIB are already available in the model library. Further libraries are "core", containing basic input/output models, and "media" which contains models related to image and video processing. The "media" repository is only partly available on Github, since it is used for student labs and that would make the solutions to the assignments available on the open internet. The components in the roof section are responsible for managing dependencies between models and components as well as repositories which can contain either model library or platform extension or both. The *Waf* build system integrates the dependency and repository managers; it is able to set up the development environment from scratch and build the actual virtual platforms. One such virtual platform is shown in Figure 3.3. The displayed platform is made up of GRLIB components and additional core components.

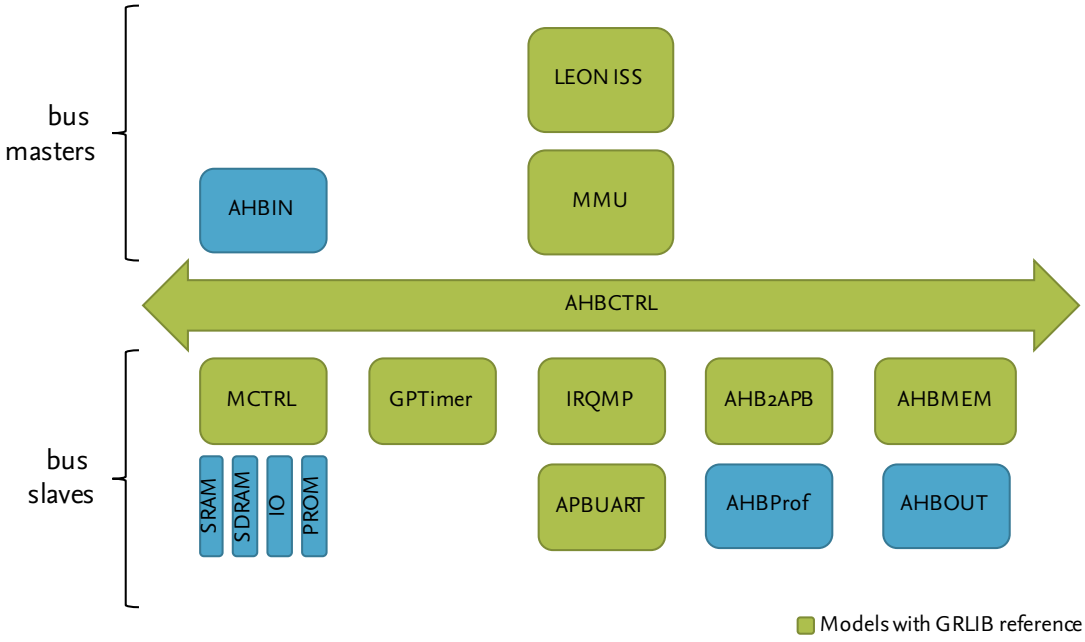

Figure 3.3: Example SoCRocket platform configuration adapted from [27]

With the recent integration of the scripting solution Universal Scripting Interface (USI), which will be explained in detail in Section 3.3.1, the framework has become much more powerful regarding its reporting and debugging capabilities [29]. Furthermore, work is underway to support dynamically creating models through a factory [30]. It is even possible to do early power estimation without the need to manually write any HDL code. USI in combination with high-level synthesis can also be used to automatically explore hardware/software partitioning [31].

Modeling Concepts

The modeling concepts behind SoCRocket have been extensively chronicled by Thomas Schuster in [32]. One of the main purposes of the framework is leveraging the development of new components and testing them on system-level before going to RTL. To make this as easy as possible, SoCRocket provides a set of library base classes, encapsulating the

communication and the storage part of the models. All models included in the "core" and "gaisler" model libraries are structured this way.

On the left side of Figure 3.4 you see a typical Advanced High-performance Bus (AHB) master component. The model is a C++ class which inherits a register file and an Advanced Peripheral Bus (APB) slave socket from class `apb_slave` and an AHB master socket from class `ahb_master`. The user just has to fill out the behavior, which is the green part in the middle. For access to the sockets a simple read/write Application Programming Interface (API) exists. The register file can be equipped with callbacks for read and write operations to any register or bit field. In approximately-timed mode, the port interface triggers a response function in the behavior as soon as data is available at the socket.

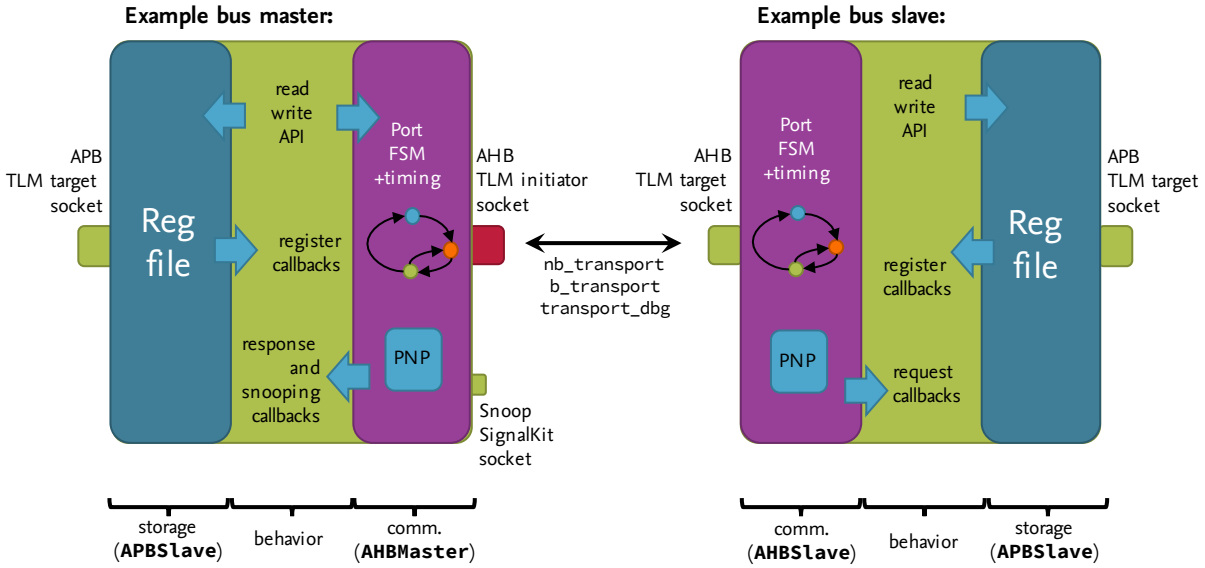

Figure 3.4: Modeling of SoCRocket models as presented in [33]

Slaves are structured in a similar way, though the implementing module now inherits from class `ahb_slave`, giving it an AHB TLM target socket. For any request, the slave interface triggers a callback in the behavior. All details of the state machines in the front-end are hidden.

If these basic concepts are followed with all components, extensions such as serialization or fault-injection facilities can be included in the base classes and they become automatically available in all current models.

Universal Scripting Interface (USI)

USI is designed as an interactive layer between a scripting language and SystemC simulation models. Initially, it was based on *GreenScript* [34], but later it was reworked to be more versatile. While *GreenScript* is aiming to integrate high-level Python models into SystemC simulations for the purpose of efficient co-simulation, USI's focus lies with interaction of existing SystemC models and modification of simulation configuration parameters during runtime. Furthermore, USI is not limited to Python as a scripting language. The *interface delegation* shown in Figure 3.5 can be adapted to any scripting language. Currently, working implementations exist for Python, Ruby, and TCL. In the following paragraphs, I will give an overview about the technical details of USI.

Basic simulation control is achieved through SWIG [35], which can automatically wrap base classes into usable APIs for scripting languages. With this baseline, it is possible to start, stop, and pause SystemC simulations during runtime from a scripting language interpreter. By using system-signal handling as well as introspection, the interface can be used to load a script before starting a simulation or to drop into an interactive shell when the simulation is interrupted. Hooking into all phases of the SystemC simulation is made possible through the registration of callback functions in every phase. USI introduces new phases to simplify initial preparation and stopping of the simulation: `{ start | end | pause }_of_{ initialization | elaboration | simulation | evaluation }`. The `pause_of_simulation` phase is used to start an interactive shell that can be toggled with the standard interrupt signal.

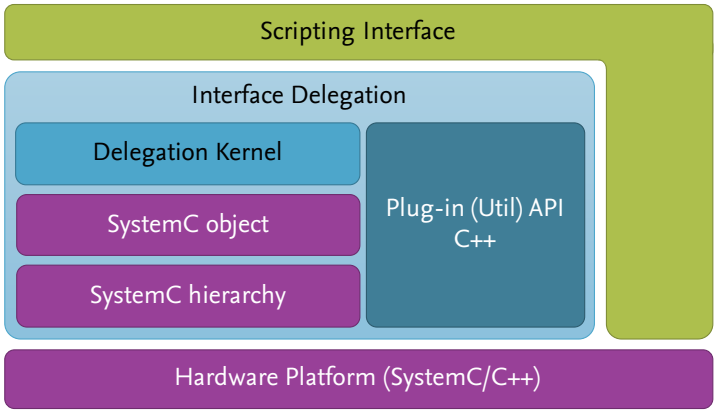

Figure 3.5: USI environment adapted from [27]

Building on these basic features that can be used with most scripting languages, USI offers further interaction capabilities with third-party components and APIs, as for example *Configuration, Control & Inspection (CCI)* or *Cadence scireg* [36]. These advanced capabilities are achieved through the aforementioned *interface delegation*. This enables developers to integrate their own debugging tools into a scripting environment.

Figure 3.5 shows the environment in which USI operates. The previous paragraphs have described the scripting interface. The *interface delegation* creates an abstraction on top of the *SystemC hierarchy* and the *SystemC object* base class `sc_object`. The delegation object aggregates interface implementations in the *delegation kernel* and creates usable SWIG object proxies for the *scripting interface*. This enables scripting language independence as described in [27]. The *Plug-in API* allows third-party tools to implement functions on top of every `sc_object` base class addressable by their SystemC name.

3.4 Continuous Integration for Hardware Design

CI describes a methodology of software development. Its goal is to enhance software quality and developer productivity. It is not an isolated step in the development cycle, but a continuous process during which single components are combined into an application.

Usually this methodology is comprised of several ingredients:

- Rebuilding the whole system

- Automated tests
- Metrics for measuring quality

Due to the progressive character of CI, it lends itself to the integration using a version control system. This way the methodology is directly interleaved with the regular development process. A developer does not even have to learn any new steps in his development cycle.

The methodology is based on a concept described by Kent Beck in his book "*Extreme Programming Explained. Embrace Change.*" [37]. Martin Fowler, an expert on agile software development, illustrates the process for a group of employees working on a common project [38]. The team members integrate their work often, usually several times a day, into the main project. Every integration is verified by automatic builds and tests to point out integration errors as early as possible.

Martin Fowler further introduces eleven principles of Continuous Integration:

1. **Sharing a common codebase**

   Every team member works on the same codebase. This way the divergence of work is avoided and the continuous integration steps are simplified. A version control system like *Git* or *Subversion* keeps track of everything. Even if the version control system supports working on multiple branches, this feature should be used sparsely to not promote divergence.

2. **Automating the build process**

   The building of a complex software project often involves several configuration tasks and compiler calls. These should be integrated into a build system like *make* or *waf*. Every developer should be able to build the project just with access to the codebase and the build tool.

3. **Executing self-tests**

   The automated build of the project does not protect from bugs in the code. Logic errors are not covered by language-intrinsic protection mechanisms. These kinds of errors have to be found through testing. Tests are also a good target for automation. When tests are required early in the development process, *test driven development* can be implemented. The test quality can be verified through *code coverage analysis*. Should any self-test fail, the integration has to stop and inform the developer.

4. **Continuous code delivery**

   The participating developers should commit code as often as possible to the common code base. This practice ensures that very minimal changes exist between code revisions. Should an error occur during integration, it will be much easier and faster to spot.

5. **Continuous codebase integration**

   The automated self-tests cover lots of error classes. Although, testing on various platforms is not covered yet. Different platforms can lead to different errors in the build process or execution. This can be done in two ways: either manually through a

developer running the integration test on the target platform or, preferably, with the help of a continuous integration server that supervises the codebase and runs tests automatically on multiple target platforms.

6. **Immediate error correction**

   To ensure a high level of software quality, the codebase should always maintain a stable version. If an error occurs, it could lead to developers working with an unstable version and introducing follow-up errors. Therefore, error detection has a very high priority. Debugging usually takes place on a dedicated development platform and not the integration platform.

7. **Accelerating build and test**

   CI requires quick feedback about the state of the codebase. Usually, building and especially testing can take quite some time. This time can be reduced by ordering the tests by priority. Tests that target basic functionality get the highest priority. Maintaining a good balance between fast tests and code coverage is the goal here.

8. **Testing in the target environment**

   The goal of the aforementioned tests should be to minimize the risk of encountering errors on production systems. Hence, it is mandatory to maintain a testing platform that resembles the target environment as closely as possible.

9. **Availability of a working version**

   A current, running version of the product should always be available to all developers in a central location. This way the current state can be assessed and compared to expectations. It is good practice to not only keep the latest version but also the previous milestones.

10. **Constant communication and feedback**

    During practice of the CI methodology it is imperative to keep a constant overview of the projects state and its current changes. For this purpose, systems that keep track who changed what including corresponding test results are employed. In most CI frameworks, these systems take the form of a webpage that also displays overall progress of the project.

11. **Automated product delivery**

    At the end of the software development cycle stands the product delivery to the end-user. To reduce the risk of error at this stage and to avoid giving more manual tasks to developers, this step should also be automated. The delivery can also include a rollback mechanism to quickly supply the end-user with the last working version, should an error occur during usage of the software product.

The most critical aspect of the CI methodology is the test quality. Therefore, special attention should be given to their development. Although Martin Fowler formulated quite aptly: „*Imperfect tests, run frequently, are much better than perfect tests that are never written at all.*“ [38] Creating tests manually can take much time and developer effort. The developer has

to fully understand specifications and interpret them to formulate tests, but a developers creativity in devising tests can only go so far. It makes sense to also automate the test generation process through randomness and fuzzing. Developers set some limits and rules for the test generation system and then automatically generate a large number of tests that test all sort of behaviour. Fuzzing has become very popular in the security-research community and has been proven very effective in finding bugs.

Having automatically generated tests is one thing. The other thing is having a test system that can support all kinds of faults: unexpected abnormal states, expected abnormal states, expected faults, unexpected faults, and expected operational states. Especially when developing hardware and software for embedded systems testing for unexpected faults becomes quite difficult, since it is hard to replicate the exact environment in which the final product will be used.

This is where virtual platforms can shine. They make it possible to integrate hardware development into a software-development methodology like CI. Virtual platforms offer the simulation speed and the testing capabilities that are required to fully support the underlying principles of CI [39].

3.5 Universal Verification Methodology

In the previous Section, I have explained the concept of *Continuous Integration*. Testing is a crucial part of CI. Without an efficient testing methodology and an easy way for developers and designers to write tests, the whole CI process cannot be established.

In hardware design, meticulously testing procedures have tradition, since creating faulty chip designs will be very costly. In software design, testing gets often times neglected due to budget issues or just plain reluctance on the developer side to write tests. In software, bugs can just be fixed with the next version update anyway. With concepts such as CI or *Test Driven Development*, the mindset is changing and more developers are at least writing unit tests.

In ESLD the hardware mindset meets the software toolkit. Universal Verification Methodology is well established in lower abstraction levels than system or transaction level (see also Figure 3.1). Fortunately, it is flexible enough that it can also be used to test on the system or transaction level. UVM offers a powerful toolkit to establish efficient testing of models at multiple abstraction levels and implementations in various languages. This makes it a very good candidate for the testing part of a Continuous Integration workflow for virtual platforms.

Even though the main focus of this thesis will be SystemC, some UVM fundamentals have to be explained, particularly as UVM also uses TLM, but not the same that is used in SystemC/TLM.

UVM has been in development for a decade and has recently made it into an IEEE standard. It is available as IEEE draft standard 1800.2-2017 [40, 41]. The default UVM uses SystemVerilog as base language. For this thesis, I chose the fairly new UVM-SystemC library. This way we do not need to delve into co-simulation and can use UVM directly in our SystemC simulation environment without the need for any external tools. In the following Sections, UVM always stands for the SystemC class library implementation.

3.5.1 Generic UVM Structure

The UVM class library, which builds upon SystemC, comes with several functions that help save time during the development of complex test benches. One of these functions is the component hierarchy, which makes it possible to create complex configurations of modules and test components with just a few lines of code.

Furthermore, UVM comes with its own configuration database which is a tool for configuring components and even exchanging them without the need to trace these changes over many positions in the source code of a test bench. Instantiation of components can be handled with the built-in factory to fully make use of the configuration database. The factory only works with previously registered data types. With the help of the factory and configuration database, it is possible to group components and instantiate them hierarchically.

This procedure is supported by the introduction of several phases that are handled automatically by the UVM library.

Figure 3.6 shows the general structure of a test bench. This is an example setup which might differ from reality. In the following Section, I will give a short overview of several components and their functionality.

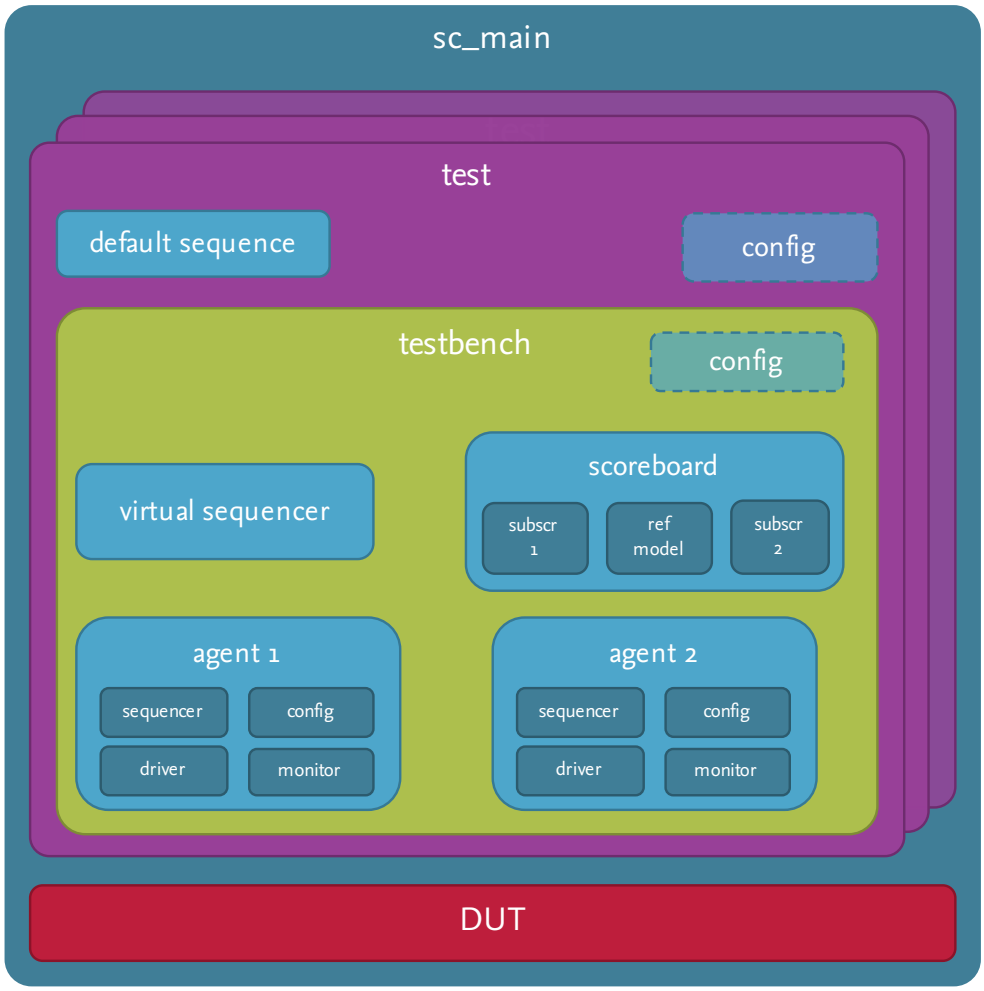

Figure 3.6: Generic UVM test bench as seen in [42]

test

The UVM *test* occupies the highest hierarchy level. Separated from access to the Device under Test (DUT), this component instantiates and configures a test case, meaning it instantiates the *test bench*. Configuration also entails selection and injection of test stimuli in the form of sequences. As mentioned before, *test* cannot directly access the DUT. Hence, this component is merely providing means for configuration and is not executing the actual test. This fact allows for reuse of many test components with multiple DUT. The components that lie below test in the hierarchy gain (with some exceptions) no direct access to the DUT as well. They describe the test scenario in an abstract way. A small number of components is responsible for the actual execution of the test.

Individual tests usually inherit from a base test that is supplying a default configuration. The derived tests can then specialize their configurations and use different sequences as stimuli.

test bench

One hierarchy level below the test resides the *test bench*. In some systems, the term test bench is used as enclosure for the testing portion of an architecture. In this sense, the DUT would be connected with the test bench during the execution of a test. Within UVM, the *test bench* has a different rank in the hierarchy. It is more the component on a high hierarchy level that is responsible for instantiating the test environment and connection of subcomponents. Keep in mind here that the test is instantiated dynamically during runtime of the simulation program. This has the advantage that the *test bench* has to be compiled only once. Hence, for many different tests the compilation of just one `sc_main` is sufficient.

scoreboard

The *scoreboard* receives information about the inputs and outputs of the DUT via an *agent* (see below). With this information, it is able to verify the behaviour of the tested module. This verification or validation usually happens with the help of a reference model or predictor, which will predict the corresponding outputs to the given input stimuli. When there is a mismatch between the DUT and reference output, an error can be issued. The implementation of this verification mechanism is not predefined and can therefore be implemented according to the specific application.

agent

The *agent* is designed as a hierarchical grouping component. It encompasses all components that are necessary for communication with the DUT. This includes the *driver* as well as the *monitor*. The *driver* is usually accompanied by a *sequencer* so that it has access to the stimuli it needs to write. The configuration database can be used to set an *agent* into active or passive mode. In active mode, a *driver* and *sequencer* will be instantiated. In passive mode, these two components will be omitted.

sequencer

The *sequencer* forwards stimuli to the *driver*. The stimuli are generated from the sequences, which contain *sequence items*. The *sequencer* decides which *sequence item* to process next, thereby functioning like an arbiter for multiple competing sequences.

sequence
The *sequence* is closely connected to the *sequencer*. Each *sequence* is assigned to one *sequencer* for it to be executed. Unlike the *sequencer*, the *sequence* is not part of the component hierarchy since it has no influence on the structure of the test architecture. Moreover, it contains information about the test procedure. This information is stored as *sequence items*, which usually contain transactions. *Sequences* can be self contained or combined hierarchically where a parent sequence can call a child sequence.

sequence items
The UVM library facilitates communication between components using transactions. Abstraction is given through TLM which allows for component reuse. The *sequence items* fulfill the role of transactions. For specific tests, the *sequence items* are derived from a predefined base class.

driver
The *driver* receives the aforementioned transactions from the *sequencer*. This data might need to be transformed through several abstraction levels until the *driver* can send it directly to the DUT. Hence, the *driver* is an interface between the UVM class library and the module under test.

monitor
The *monitor* is another interface like the *driver*. This passive interface registers the outputs from the DUT and transforms them, analogous to the *driver*, over several abstraction levels into transactions. These transactions are then sent through a broadcast interface so that multiple test bench components can receive and process them.

environment
The *environment* provides solely grouping functionality for components. An *environment* can also contain other environments, although usually *agents* and *scoreboards* are instantiated within an *environment* so that one *environment* belongs to one component under test. Like with other encapsulating components, the usage it optional.

3.6 Summary

In this Chapter, I explained the fundamental concepts that need to be understood to grasp the subsequent Chapters. First, I clarified the terminology with definitions of the terms *checkpoint* and *snapshot*. Then, I explained several further terms from the checkpointing domain. This I followed with a short overview of serialization. Since the thesis topic is snapshotting for SystemC based virtual platforms, I also explained briefly the concepts behind SystemC based virtual platforms and shed some light on the SoCRocket virtual platform framework, which I use in this thesis to implement my snapshotting framework. The SoCRocket description highlighted the modeling concepts and the universal scripting interface of that particular virtual platform. The concept of continuous integration for hardware design needed to be introduced. Until recently continuous integrations was only prevalent in the pure software domain. With virtual platforms and modern test frameworksi, the concept becomes also intering in hardware design. With UVM the de-facto industry standard for testing and verification of hardware models was introduced in this Chapter as

well. In the later Chapters, it will be used to implement example use cases for the SystemC snapshotting framework I present in this thesis. My snapshotting framework combined with the UVM library makes the SoCRocket virtual platform ready for hardware and software development in a continuous integration workflow.

In the next Chapter, I will look in detail into the current state of the art of snappointing. The different approaches for snapshotting will be distinguished in a historic context. Furthermore, I will especially look at what has been done in the SystemC domain with respect to snapshotting until now.

4 State of the Art

In the previous Chapter, I gave an overview of the fundamentals needed to understand the thesis. In this Chapter, I will give an overview about the state of the art with respect to checkpointing mechanisms in general and their historic context. A list of improvements introduced with the C++11 language standard follows. The next Section will show several current serialization solutions that could form the basis for data storage for a checkpointing implementation. The penultimate Section focusses on applying the continuous integration concept in the embedded systems domain. Finally, a summary of the related work can be found in the last Section.

4.1 Checkpointing Mechanisms and their Implementation

The origins of snapshotting were with mainframe systems and early distributed clusters. These machines were not fault-free and to ensure feasibility of long running simulation various snapshotting techniques were employed, albeit at that time they were called rollback and recovery. When an error during computation was detected the simulation would be stopped and restarted from the last known-good snapshot. This error recovery from a saved state can also be useful during debugging of systems and applications. Since storage was still expensive, creating periodic snapshot with high frequency was not an option. Lots of work went into optimizing the amount of checkpoints needed to be saved depending on the application. Nowadays, storage is fast and cheap and many more snapshotting and checkpointing mechanism have been developed.

Most people think of virtual machines when they hear the terms checkpoint or snapshot. Many don't realize that they have probably played around with snapshotting and checkpointing before they even knew what a virtual machine is. Video game save games are a very ubiquitous incarnation of classic snapshotting. The current state of the game world is saved and later restored when the game is resumed. Depending on the style of game these save games can be created periodically, at certain key points in the game or completely manually by the player. This is completely analogous to how checkpointing works in distributed systems or virtual machines. Unfortunately, there are not many publication about how save games are implemented in video games, since almost all implementations are proprietary. Furthermore, judging by the short development cycles there probably is not enough time to write pretty code and document it, let alone write a publication about it.

The fundamentals of checkpointing and snapshotting have been explained in 3.1. The related work collected here will be broken up into the different abstraction levels they address.

4.1.1 System Level

At the system level, we can start with the rollback-recovery mechanism that have been around for some time. As mentioned before, these were mainly used in distributed systems in the past. Many researchers started their work at the IBM T.J. Watson Research Center and continued their research elsewhere.

In [43] Chandy describes how systems made up of unreliable components can be made reliable through the introduction of redundancy. This redundancy can be in hardware, software or both. He describes two example systems, a database with many users and a process control system without much interactivity. A database system can have a permanent record of all transactions for auditing purposes. This already helps with reliability, as the record can be used to recover a system state by playing back the transactions. Additionally, at certain points in time a snapshot of the whole database can be saved and archived, but this process might take much time. During the snapshotting process no transaction are allowed. The storage of transaction records and database snapshots imbue a cost for the reliability they provide. This cost can be material (storage) or immaterial (downtime). Chandy provides an analysis of the optimal times to take snapshots of the two example systems. For the process control system the optimization requires a programmer to analyse the tasks and estimate execution times. This analysis by itself might be too expensive. Overall, Chandy lays the theoretical background for further works.

One such work by another IBM fellow is found in [44]. Randell gives a more practical guide to designing fault tolerant software systems. He postulates that software faults generally stem from design errors, since software design is much more complex than hardware design. The difference in complexity is reasoned to result from the amount of internal states possible in software compared to hardware. The software structure he proposes consists of a multi-level system design. At the lowest abstraction-level is the processor, on top of which the IO system sits, then a device layer, followed by two file system layers and file access methods, topped by the actual user program. All these levels constitute separate "virtual machines" that communicate through internal interfaces between the abstraction layers. Each virtual machine provides atomic operations. The fault tolerance in this multi-level system is implemented in the interfaces between the virtual machines. This way much complexity is abstracted away from a programmer and he has only safe function at his disposal when designing software at the highest abstraction level.

Koo and Toueg [45] built on the theoretical groundwork from Chandy and Randell. They expand the algorithmic theory to distributed systems, but explicitly exclude database systems from their treatment. Their focus lies on maintaining a consistent system state during checkpointing and rollback-recovery operations. The distributed systems for which their algorithm can be applied shall have the following characteristics: no shared memory between processes and communication through message channels; channels are virtually lossless and first-in-first-out; process can fail and other processes are informed about failure. Having processes create independent snapshots can lead to a "domino effect". The approach chosen by Koo and Toueg involves a process coordinating the other processes during the checkpointing phase. The same observation is applied to the rollback-recovery mechanism. Their algorithm uses a two-phase-commit protocol. In the first phase, only tentative snapshots are created. In the second phase, the initiating process decides if theses tentative snapshots should be made permanent by checking if all processes succeeded in taking tentative snapshots. Since either all or none of the processes save a permanent snapshot, the most recent set of checkpoints is considered consistent. The initiator process can be a special "daemon" process that supervises the system. The rollback-recovery algorithm works in the same way.

In [46] Israel and Morris build upon the previously mentioned works and describe a non-

intrusive checkpointing protocol. For their analysis they handily use the same terminology as Chandy in [43]. Their focus is also on maintaining a globally consistent state within a distributed system, albeit with as little interference with normal system operation as possible. The authors suggest it is beneficial to create periodic system snapshots, although they don't specify the interval. The described checkpointing mechanism works in a very similar way to the one describe above by Koo and Toueg. Although they remove some restrictions. The processing does not need to interrupt normal operation during the creation of tentative snapshots and tentative snapshots are not discarded if another process fails during the creation phase.

Another approach is to model concurrent checkpointing and recovery as a transaction processing problems. Leu and Bhargave show in [47] that this way the checkpointing and recovery of multiple processes can be solved by enforcing serializability on the modelled transactions. In a distributed system where processes communicate by exchanging messages, these messages usually trigger actions. The problem of taking a snapshot of a distributed system is dealt with through the synchronization of snapshot operations, messaging operations as well as rollback operations to ensure a consistent overall state. The authors refer to the previous works described above as a synchronous approach which they modelled as a concurrent transaction processing system. In this model, each send message is coupled with the corresponding receive message to form a transaction. These transactions can then be synchronized using a locking protocol devised by the authors and subsequently serialized for storage. This is one of the earlier works were serialization is mentioned in conjunction with snapshotting.

While the aforementioned works were largely of a theoretical nature, Elnozahy et al. give an exhaustive survey of rollback-recovery protocols in message-passing systems in [12]. Not only do they list and explain a broad array of protocols, they also evaluate their implementations with regards to practicality. In their survey, they concentrate on fundamental concepts and implementation issues of checkpointing protocols in distributed systems. They exclude several application fields such as hardware-level checkpointing, debugging or techniques that require special language constructs. These exclusions are covered elsewhere already. One issue not covered by the other mentioned works is the interaction of processes with the outside world. Regenerating the outside influence on a system during rollback-recovery is not trivial. The authors cover several logging protocols that support processes in saving their input messages. These protocols follow the idea of the "audit log" mentioned in [43]. Most commercial implementations of message logging use the pessimistic logging variant since recovery is simplified. In their survey, they identify access to stable storage as a major overhead for snapshotting or message logging systems. With the recent proliferation of faster storage techniques this should not be an issue any more. The authors also include a brief distinction between system-level and user-level checkpointing to which we will come later. Furthermore, the survey covers the aspect of checkpoint frequency and placement. Regarding practicality of the surveyed checkpointing protocols, Elnozahy et al. note that mostly scientific simulations running on large scale distributed systems benefit from them which could explain while they are not widely adopted. Although, with the advent of cloud computing and the ubiquitous usage of virtual machines these protocols might experience a renaissance.

In a more recent work [48], Hernandez and Abella present low-cost checkpointing for

automotive safety-relevant embedded systems. In multi-processor systems, running mixed-criticality tasks, resetting the system to recover from a fault is not an option, since it could affect critical tasks. They take the theoretic algorithms and analytical approach from Chandy and apply them embedded software. They test their implementation on the Infineon AURIX platform in the form of a SystemC model. As virtual platform they use a modified version of SoCLib [49], which has been in development for over 10 years now and looks like it is not in active development any more but receives occasional bug fixes. They have used this rather outdated virtual platform as it offers the ability to inject faults into the simulation components, which is not possible with commercial tools. The authors evaluate several checkpoint and recovery mechanism with regards to overhead. They specify checkpoints to only be possible at certain points in the execution of tasks. Furthermore, their simulated tri-core system runs in light lock step to maintain redundancy if necessary. The authors solution enables scheduling of task checkpointing while maintaining safety relevant guarantees.

Sun Microsystems (now Oracle) has at least two patents on checkpointing methods [50, 51]. In their patents they reference the works from Chandy, Koo and Elnozahy. The patent text is more difficult to decipher than a scientific paper, but it seems as if they implemented the theoretic algorithms described before in their distributed multi threading system. One patent is more focused on the system level while the other also goes into detail about the application-level implementation. Their approach uses code injection to augment existing code with checkpointing functionality. Oracle also maintains VirtualBox, a popular open source virtualisation solution. In the next Section, we will have a closer look into virtual-machine-level checkpointing.

Summary

Checkpointing at the system level is mostly interesting from an historic perspective to learn about the theory of checkpointing and established methodologies. Nowadays, most checkpointing implementations are at higher abstraction levels. The early works [43, 44] focus on IBM mainframes and distributed systems with the main goal of keeping the systems reliable and recover quickly from faults. Interestingly enough Randell introduced the virtual machine concept in [44] and proclaimed that software is more complex than hardware. Today, we can clearly see that both areas are becoming more complex in an upwards spiral of complexity.

Other works such as [45] and [46] went on to focus on non-intrusive ways to implement checkpointing for distribute systems.

Then in [47] the concept of transactions is mentioned. The application was still for distributed systems, but now the transactions passed between models was the target of checkpointing. In this work, serialization was introduced also. The theoretic concepts introduced with this work helped shape the design of my own snapshotting framework.

The survey in [12] covers recovery of message passing systems, which are conceptually quite close to SystemC/TLM simulation models. Further topics in the survey were the interaction of systems with their environment and the impact on checkpointing. The environment in this case is considered to consist of storage or networking peripherals, which shall also be covered by the checkpointing process.

In [48] the authors used SystemC to evaluate a system-level checkpointing methodology on an embedded system. The interesting take back here is that they used SystemC mainly to inject error into the simulated embedded system to exercise the recovery mechanism of

their checkpointing implementation. They used a rather old SystemC simulation framework which shows that there is still a need for a modern SystemC framework with fault injection and snapshotting functionality.

Oracle, the current maintainer of the VirtualBox virtual machine software, has several patents related to checkpointing. In their patent texts they reference the same theoretic works as I have done above. Their checkpointing technique involves augmenting existing code through code injection with checkpointing functionality. The idea is quite interesting, but difficult to implement at higher abstraction levels.

4.1.2 Virtual Machine Level

One of the bigger research areas for checkpointing is virtual machine checkpointing. Traditionally this is done with stop-and-copy checkpointing, where Virtual Machine (VM) execution is stopped during the checkpointing procedure. This is already implemented in hypervisors like KVM [14] and Xen [52], but causes disruptions of interactive services and can lead to long delays.

SimOS [53] can be considered an early precursor of QEMU. It was developed as a simulation environment for operating systems. The simulator runs completely in the user space without any hypervisor support. It is difficult to clearly classify SimOS between application level and virtual-machine level. The authors see it more as a virtual machine than a typical application so I have included it in this Section. Similar to QEMU it uses existing operating system facilities like processes to simulate hardware components. The hardware components are abstracted enough to allow execution of an OS and have good performance. According to Rosenblum and Varadarajan SimOS can reach between 50% and 100% of native execution speed. SimOS offers several interfaces to the state of the simulated machine. One interface allows a remote debugger to connect directly to the simulated OS. Another interface, called the detailed simulator interface, enables a developer to encapsulate the entire state of the simulated machine. This encapsulation can be used to have two simulation modes, fast and detailed. Furthermore, it offers the ability to save the state of the simulation to disk. A checkpoint file will contain the register and memory state of the simulated machine as well as the state of connected I/O devices. To reduce checkpoint file size only changed blocks of disk I/O devices.

Brendan Cully gives a good general overview of VM checkpointing as well as some more details on Xen in [54]. As motivation for checkpointing he gives recoverable long-running processes, time-travelling debugging and forensics. Recoverable long-running processes were also the motivation behind system-level checkpointing. Time-travelling debugging will be explained later in more detail. Forensics is helpful in malware research or in general to find out what caused failure or otherwise unwanted system behaviour after the fact. At the time of Cullys presentation, saving the state of a Xen VM still needed to pause the VM for some time to save memory and device state. Later Xen got its own copy-on-write implementation and checkpointing a system while it was running was possible. The Xen hypervisor also enlists the guest-operating-system kernel in supporting the snapshotting process. The guest kernel should be able pause connected devices and enable shadow mode for the memory, so pages can be copied. The format for the saved snapshots is streamable, which enables live migration of VMs from one domain to the other by simply piping the save output to the restore input on the other side.

Ta-Shma et al. use Xens live migration feature in their work on VM time travel [55]. To

enable time travel with both transient and persistent VM state they combine checkpointing with continuous data protection (CDP). Continuous data protection is a storage technology that enables restoring a storage devices state to a previous point in time by keeping a history of modifications. Typically, VM checkpointing does not address external storage devices. For the time travel support VM snapshots are synchronized with the CDP history. Their implementation does not require special support from the guest operating system. Since this work happened in parallel to Cullys work on Xen checkpointing, Ta-Shma et al. developed their own checkpointing extensions for Xen. They wrote prototypes in Python and the final implementation was done in C. Their checkpointer intercepts live migration bitstreams and saves them to disk. Upon restore the checkpointer poses as the migration target VM and plays back the live migration stream to the original VM. The live migration data contains the transient state of the VM. The persistent state of the VM is saved in the CDP storage device. Synchronization of both states happens during the short time during live migration where the VM needs to be paused to get a consistent view of VM memory. The authors do not mention explicitly which file system they are using for their storage back-end, but they mention that ZFS [18] could be used with a high snapshot frequency as it supports large amounts of snapshots.

Some people position virtual-machine-level checkpointing as a replacement for process-level checkpointing. In [56] Liu et al. argue that virtual-machine-level checkpointing offers the advantage of compatibility, transparency, flexibility and simplicity. However, the checkpoint size at this level is an issue if the goal is to just have a minimal system in the VM to save the state of one process. Their approach is to use memory exclusion with a ballooning mechanism. This mechanism will not save unnecessary free pages thereby reducing the amount of memory being snapshotted. The user-level checkpointing library Libckpt [10] uses the same scheme. The authors implemented their ballooning mechanism inside the kernel space of the guest as a driver that communicates with the Xen VM manager. The memory balloon takes up free memory in the guest, so that the checkpointing routine does not access it. Reducing the checkpoint file size reduces the time it takes to write the checkpoint as well as the time to restore from a checkpoint. Depending on the application running in the guest, savings in time and checkpoint size can be quite significant. Although, when running a memory intensive application the checkpoint size reduction becomes negligible. This technique is therefore only suited for very specific applications.

Another promising approach for reducing the size of VM checkpoint images is presented by Park et al. in [57]. Their approach is to only store memory pages in the checkpoint image file that are not already available in persistent storage. In a modern operating system, the kernel allocates memory that is not used by itself or running processes to the so called page cache. This cache contains data recently read or written from block devices in order to reduce access delays, when they are needed again. In the Xen version the authors work with ballooning is already included in the VM manager. Finding out which parts of the page cache are available in persistent storage requires either modification of the guest operating system or the VM manager. The authors have evaluated both approaches. For a fully virtualised guest they achieve a reduction in disk space of 64% and reduction in time of 62%. They even manage to outperform the incremental checkpointing approach in disk space and time by 57% and 26% respectively.

Siripoonya and Chanchito have developed Thread-based Live Checkpointing (TLC) [58],

which as the name suggests TLC is designed to do live checkpointing of a running VM. This is enabled through multi-threaded CPUs. Besides the VM thread a new thread named checkpoint thread is created. This thread is responsible for interacting with checkpoint files. The virtualised system state is usually held within non-volatile memory. While the checkpoint thread is running, the VM can resume normal execution. It is regularly interrupted when the checkpoint thread needs to copy dirt pages to its hash table. For this TLC needs extra memory. After the checkpoint thread has finished, the checkpoint is written to disk. The TLC mechanism comprises four phases: 1. Create data structures in memory (hash tables, bit vectors for dirty pages) and create checkpoint file. 2. Write all memory pages to checkpoint file. At the same time the VM resumes and is interrupted occasionally to write out dirty pages. 3. Checkpoint thread signals to the VM thread that it has finished writing the memory contents to disk. This triggers a last write of dirt pages followed by saving device states and disk states. 4. The hash table is written to the checkpoint file. Experiments by the authors have shown an increased checkpoint performance by a factor of 0.53 compared to traditional stop-and-copy checkpointing. Although, TLC requires more computing power and memory resources.

The prevalence of cloud computing enables researcher to run distributes application on an array of networked virtual machines. Typical applications include MATLAB or IPython [59] which have integrated support for parallel computation. As with older rollback-recovery schemes, researchers still want to take snapshots of running simulation or computations. Garg et al. have developed a checkpoint restart solution for a network of virtual machines [60]. DMTCP is a common user space library for checkpointing distributed applications and will be described in detail in the next Section. It is extensible through a plug-in interface. Since KVM/QEMU [61] virtual machines are basically applications Garg et al. are using DMTCP to checkpoint their networked VMs. With user-space QEMU it worked right out of the box without any modifications on QEMU or DMTCP. With the kernel hypervisor DMTCP needed to be extended with two plug-ins to support the checkpointing of a VM and its network state. As storage back end the authors used the BTRFS [19] file system as it has direct checkpointing support and a raw image can be used from the VM and the host. In their experiments they achieved sub second checkpointing times.

Fault recovery is an important feature for mixed-criticality embedded systems. Sair and Psarakis have ported the Xen hypervisor to an ARM based embedded system to improve reliability and isolation of embedded applications [62]. They tested their approach with a system running two Linux VMs and one FreeRTOS VM. They evaluated different checkpointing approaches: Application level within the linux guests using the Berkeley Lab Checkpoint/Restart (BLCR) library [16]: Application level within FreeRTOS using own checkpointing code; Combined application-system-level with support from the hypervisor; Pure virtual-machine-level checkpointing using the hypervisor as only checkpointing manager. The result of evaluating the different checkpointing schemes is that while the virtualisation improves reliability it comes with a significant time overhead. This overhead might interfere with the criticality of the running applications.

Summary

Implementing checkpointing at the virtual-machine level is the most prevalent implementation right now, but it has two obvious drawbacks. The first being the checkpoint size and the second the overhead of having to simulate a whole operating system to maybe just

checkpoint one single application running inside of it.

In [53] the authors present a simulation framework which behaves very similar to a SystemC/TLM simulation framework in that it has two abstraction levels it can run in and that it runs completely in user space. Their checkpointing methodology covers registers and memory as well as the changed blocks of connected storage devices. Focussing on the registers and memory of the simulated system is a very good point that I will take up later in my design. The drawback of their work is the intrusive implementation, which can only be done when the whole simulation framework is under their control. In my case, I would need to upstream changes to the SystemC simulation kernel.

The motivation in [54] is checkpointing of long-running processes, time-travel debugging as well as forensics. Except for forensics I have the same goals with my snapshotting framework. Cully focuses solely on the Xen hypervisor and chose an intrusive implementation method for his checkpointing. It needs support from the guest operating system kernel, which is not an option for SystemC simulations.

The authors of [55] again focussed on the Xen hypervisors. Their goals was to be able to time travel between checkpoints. They achieved this by hooking into the live migration data stream of the Xen hypervisor and relying on functionality of the underlying storage system to create snapshots. This will unfortunately only work with a VM and is not applicable to my use case of snapshotting a SystemC/TLM simulation.

In [56] the authors used virtual-machine-level checkpointing as replacement for process-level checkpointing. They used a very minimal operating system to run their application they want to snapshot. To circumvent the large checkpoint size, they employed clever memory management techniques. The concept is interesting and could also work for SystemC/TLM simulations, but to my knowledge this has already been done by a research group in Bremen [63]. Furthermore, the checkpoint size is still quite large.

In [57] the authors addressed the VM drawback of large checkpoint size, but unfortunately their solution requires modifications to the VM manager and the OS. This is of course no option for a SystemC simulation framework where the operating system might be part of the simulation and the SystemC kernel represents the VM manager which should not be modified.

Another drawback with virtual-machine-level checkpointing is the fact that the system needs to be paused during the checkpointing process. This was addressed in [58] by implementing a live checkpointing methods which leads to faster checkpointing times, but needs more memory resources and extra threads for the checkpointing process. Having an extra thread to handle the checkpointing while the rest of the system keeps running should also work inside a SystemC simulation since the SystemC kernel does all the thread handling.

Distributed systems were mostly covered with system-level checkpointing, but the authors of [60] implemented a checkpointing methodology for distributed virtual machines. They used the user-level checkpointing tool DMTCP, which will be covered in the next Section, to checkpoint the VM processes running in user space. They also used storage system features for checkpointing as has been done in other works. Using DMTCP seems like an interesting approach and will be investigated further in the following Sections. Reliance on storage system features is a drawback. Another good point to take from this work is the way they extended existing frameworks to implement their checkpointing methodology. This is also the way I will go with my snapshotting implementation.

Another work from the automotive domain was presented with [62]. The authors goal was to introduce fault recovery in a mixed criticality system. They ported the Xen hypervisor to the ARM architecture to use it for checkpointing. The virtualisation approach improves reliability but is ultimately not usable for critical applications due to the large overhead. They give an interesting overview of the whole checkpointing process, but the solution does not really fit my use case.

Overall, there are many interesting concepts in virtual-machine-level checkpointing. However, the drawbacks overshadow the positive sides of this level of checkpointing implementation, which is why I will only take some of the concept and see how I can apply them to SystemC checkpointing. The user-level checkpointing implementations which I will cover in the next Section look much more promising.

4.1.3 User Level

There are also several checkpointing implementations operating at the user level. As long as programs do not rely too much on kernel data structures, e.g. file descriptors or message buffers, checkpointing at user level is a viable option. Since most software that requires long execution times and would benefit from checkpoint-restart mechanisms is of a scientific nature and involves more computations than file operations, this should not be an issue. Usually, the developer needs only to change very little in his code and link to the checkpointing library to benefit from its features.

One of the more well-known user-level checkpointing libraries is *libckpt* which is presented by Plank et al. in [10]. They have developed the library to support developers that could benefit from checkpointing and recovery, but don't want to implement their own custom solution. To use the library a developer just need to rename his `main()` function and link his code with the `libckpt.a` library. The library works under many flavours of Unix. Several optional checkpointing features are available in the library. The user can choose to enable incremental checkpointing, which reduces the size of subsequent checkpoints. If the process being checkpointed shall continue execution while the checkpoint is being written, the forked checkpointing can be enabled. Forked checkpointing uses the Unix `fork()` primitive to create a child process which handles the checkpointing while the parent process continues execution. Checkpoint files can optionally be compressed to save space. A more flexible solution for saving space is memory exclusion which can either be automated (during incremental checkpointing) or manually by the user who can exclude specific memory areas. Furthermore, instead of taking checkpoints at specific intervals the user can specify in the code where checkpoints should be taken by placing a special function call. The checkpoint overhead times times presented in the paper seem a bit large whereas the checkpoint image sizes seems relatively small. Furthermore, the library was not tested with multi-threaded applications.

Litzkow et al. developed process-level checkpointing in the context of a distributed batch processing system [64]. They needed the ability to quickly migrate processes from one workstation to another. Applications being run as distributed jobs are relinked with their checkpointing library. Their approach seems to be very similar to what Plank et al. describe in [10]. They furthermore list the same limitations as other user-level checkpointing libraries.

Many applications use multiple threads and should also benefit from user-level checkpointing. Dieter and Lumpp have developed a checkpointing library [11] that supports

programs using POSIX threads. As mentioned before, user-level libraries cannot access the kernel memory. This memory used by the kernel to manage threads and keep track of their state. Modifying the kernel to allow access to this memory area is not an option as it would make the kernel less secure and the checkpointing library dependant on a specific kernel version. Moreover, users usually do not have the right to install a modified kernel, whereas linking libraries to user owned applications is okay. A user-level checkpointing library supporting multithreaded applications has to live with some limitations. The library cannot support programs that randomly access files or communicate with other processes. Furthermore, the library does not support POSIX semaphores. The library uses Unix signals to synchronize application threads at the checkpoint saving time. To get access to all the necessary information to correctly save the state of several threads, the checkpointing library intercepts calls to the thread library. To restore open files the library also intercepts file open and close calls. With checkpointing enabled the application gains a 3% to 10% increase in execution time. The authors mention further that writing the checkpoint image to disk introduces a significant overhead which they plan to optimize in the future. In 2009 Ansel et al. published their distributed checkpointing solution Distributed Multi-Threaded Checkpointing (DMTCP)[65]. The code is available under an open source license. They introduce a transparent *user-level* checkpointing for applications that are distributed on multiple networked nodes. Since their solution is operating on the user-level it can easily be bundled with the checkpointed application. Many applications like complex simulation tasks have multiple phases where often the first phase is the same for multiple parameter variants. The idea behind DMTCP is to create checkpoints in a cluster and then use the checkpoints to continue work on a workstation or even laptop. It could also help with debugging long running jobs by taking a snapshot just before an error occurs.

DMTCP comprises two layers: Multi-Threaded Checkpointing (MTCP) [66] and DMTCP. MTCP transfers responsibility for checkpointing to single processes, whereas DMTCP is responsible for socket communication and other artefacts of distributed software. During the checkpointing procedure network traffic which is still in transit or in kernel buffers is written to process memory and saved inside the checkpoint image. This network data will be send again to its original recipients after restoring of the checkpoint. This implies that the network configuration was not changed and all recipients are still reachable. The global communication between the distributed processes is done via peer-to-peer networking using barriers.The authors achieved a checkpoint save time of 2 seconds on a cluster with 32 nodes and 128 distributed threads and report that adding more nodes did not affect the save time.

Summary

In [10] *libcpkt* is introduced. The library can be used to extend existing applications with checkpointing functionality. These application need to have a different main function and should not use any multithreading. This makes the library already unusable for SystemC simulations, as the main function already has a different name and the simulation kernel takes care of the thread management during the simulation.

The authors of [64] followed in the footsteps of *libckpt* to implement their checkpointing library for distributed batch processing systems. Unfortunately, their library come with the same limitations and is therefore also not usable for SystemC simulations.

User space applications only have limited access to kernel memory space where multiple

threads are usually managed. By intercepting calls to the threading library the authors of [11] implemented user-level checkpointing with multithreading support. They do not support semaphores. Furthermore, their solution as a large overhead on performance. As stated before, the SystemC kernel does the thread handling for the simulated models, so this could be interesting, as there is not much interaction with the operating system kernel. Although, the performance impact due to the big overhead is not negligible and makes this approach rather unattractive for SystemC snapshotting.

DMTCP was already used before for virtual-machine-level checkpointing. The framework presented in [65] provides checkpointing for distributed applications and relies on the work of [66] for the process-level checkpoints on the networked nodes. DMTCP has support for multiple threads and works also locally. The implementation is transparent to the user and no modifications to the checkpointed application are needed. This versatility makes it a very good candidate to compare against. In theory it should also work with SystemC simulation out of the box. We will see later in the evaluation Chapter how this goes. Of course by using the DMTCP application to checkpoint a SystemC simulation only the process information is saved, any insight into the model state is inaccessible to the user. As other checkpointing options presented here are either not developed any more or are simply not available to the public, this is the only option to compare against.

4.1.4 Application Level

Video games have been mentioned as example before. Here, they will serve as a first example for application-level checkpointing. Modern video games are usually developed using a game engine. The game engine is like a software framework for designing various video games. The engine handles all the complicated stuff and provides development and design tools for game designers and developers. This way the game developers don't have to think about how to implement a save game feature, they just use the one provided by the engine. Engines like the Unreal engine [67] or Unity (an open source variant) [68] can easily provide save game features with the help of internal serialization libraries. Since the developer is limited to use only classes provided by the engine, the engine provider has full control and knowledge of these classes.

Also old video games can benefit from snapshotting techniques through emulation. Since an emulator know the state of every system component as well as the memory contents, an exact snapshot of the game systems state can be easily saved to disk. In the past this was a very unique process for every emulator. Recently, projects like libretro [69] have begun work to consolidate unique emulator efforts within one common library. The approach of the libretro project is not unlike that of a virtual platform framework. Their goal is to have a library for every video game system (e.g. component libraries) with common functionality like video, sound or snapshotting being part of the main library (e.g. SystemC kernel, core library).

With games like Minecraft [70], where participants can create their own world, it is even possible to design simple computing systems [71] within the confines of the game world. As the game world is saved, also the state of the included computing system is saved, much like taking a snapshot of a running simulation.

Electronic design automation software from several vendors supports taking snapshots of RTL simulations since quite some time. Saving the state of an RTL simulation is much more straightforward than saving the state of a whole application. It might seems more

difficult because the simulated chip or system seems more complex, but the possible states are very limited and the simulator know the state of all signals and flip-flops. The only issue is the snapshot size for large systems. Although, depending on the snapshot format it should be a good target for any compression algorithm.

When developers are not confined to the walled garden of a game engine or similar framework, creating snapshots will be more difficult. Over the years many application-level snapshotting mechanism and libraries have been developed. Here, I will give an overview over a few of them.

Beguelin et al. have extended a C++ library for a distributed object migration environment (dome) with application-level checkpointing [13]. The dome library enables a programmer to develop applications that can be run in parallel on a distributed cluster. It follows a single program multiple data model for parallelization of the application. By implementing the checkpointing mechanism at the application abstraction-level the application as well as the checkpointing code stay portable. This method is not transparent to the user and requires extra work from the programmer. They also refine their method by instructing the preprocessor to do some of the task a programmer would need to do by hand. Since the dome library has control over all its objects, the authors extended these objects with checkpointing abilities. Normally, saving the state of an application requires saving and restoring the program counter, the stack, a subset of program variables, the communication state and the I/O state. Several of these states are not accessible with application-level checkpointing. This is not an issue though, since they can be omitted to ensure checkpoint portability. The program counter does not need to be saved since checkpoints will only occur when the `dome_checkpoint()` method is called. Saving the stack is not trivial. The preprocessor method of Begeuelin et al. introduces procedure calls which take care of it. The subset of variables simply includes the whole set of object belonging to the dome library. The communication state is irrelevant, since checkpoints will only be taking between calls to specific dome methods involving communication. For the I/O state it is sufficient to save and restore the file pointers currently being used. The checkpoint files use the External Data Representation (XDR) [72] format, a data serialization format developed by Sun and later used in the ZFS file system [18]. The checkpointing implementation uses a master task to collect checkpoint data from the distributed processes and write them to disk. The solution is completed with a failure demon that acts as a watchdog for running applications, takes checkpoints and restarts processes in case of failure. During evaluation of their checkpointing implementation the authors found it to have only a low overhead to execution time while significantly improving the fault tolerance of the whole dome library system.

A more recent application-level checkpointing implementation comes from the field of computational fluid dynamics (CFD). The authors of [73] developed a synchronous checkpoint-recover method for their simulation framework. They chose to implement the checkpointing at the application level to reduce checkpoint size as well as overhead during execution. Their CFD framework already takes periodic, synchronized snapshots to save intermediate results to disk. The authors hook their checkpointing mechanism into the existing structure for saving intermediate results and add functionality to restore the simulation state from a saved checkpoint. They achieve user transparency by limiting their implementation to the underlying framework. Any application developed with the

framework can therefore benefit from the added functionality. The authors tested their checkpoint-recover mechanism with several benchmarks on a high performance cluster. Checkpoint overhead depends heavily on the benchmark as well as the number of parallel processes and can range from 10% to 80%. Recover overhead was always below 0.03% percent and therefore negligible.

Also non-scientific applications can benefit from application-level checkpointing, even text editors. Text editors such as VIM [74] and EMACS [75] can be extended with various plug-ins to become more powerful than full-blown IDEs. Start-up times can also become quite long when loadings a large amount of extra modules and maybe restoring a sessions with many files open.

This is especially problematic for EMACS since much of its functionality exists in its own dialect of Lisp. During start-up all this Lisp code has to be initialized which can take a long time. The EMACS developers fixed this problem by writing a memory image from the initialized state to disk and use the image during the next start-up. This was implemented by basically dumping the memory and later using something called "unexec" to restore the memory from the dump. The "unexec" code [76] converts a running program into a memory dump. Unfortunately this special EMACS feature uses special Glibc functions, which were deprecated from the library in early 2016 [77]. The disabling of these functions caused Emacs to fallback to a different dumper implementation which resulted in unstable behaviour [78]. Soon afterwards a patch [79] adding 4500 lines of code to EMACS was submitted, which included a workaround for the unstable behaviour and a new implementation to create the memory snapshots. The new implementation uses low-level C code which is not favoured by EMACS developers and they worry that it will become unmaintainable in the future. Another solution would be to vastly improve the Lisp-loader performance, but no one is working on that. This issue shows the importance of not relying on archaic library functions and writing future-proof and portable code.

During ESLD software simulators play a large role. They can be used to explore the design space or test operating system and drivers. These simulators can also benefit from checkpointing like a virtual machine would. Simulation time can be saved by saving a booted operating system state when running benchmarks or driver tests. Gem5 [80] is a well-known simulator uses exclusively in research environments. It is available as open source and therefore easy to modify and extend for individual purposes. Unlike SystemC Gem5 does not aim to create accurate models, its focus is more like QEMU in providing good software performance. The Gem5 models are purely functional. The simulator has built-in support for snapshotting. The snapshotting feature is implemented as serialization of the simulation objects. Since every simulation object is derived from the same base class, it was sufficient to include save and restore methods in the base class. These methods have to be implemented by model developers if they plan to take snapshots of their simulated system.

Summary

Many ubiquitous applications come with checkpointing applications, some rather obvious ones such as game engines or emulators, and some not so obvious ones such as text editors. Game engines and emulators work in a very similar way as simulation systems. They represent complex systems made up of interacting objects. In one case, the objects can represent an environment and in the other they represent parts of an embedded system.

The mentioned applications tend to use readily available checkpointing concepts, but it is hard to analyse them in detail as not all have their code available to peruse.

An example for checkpointing in a text editor was available until recently with Emacs, but the *unexec* feature had to be removed due to changes in the underlying libc. EMACS relied on obscure library features to implement *unexec* which have been deprecated. This is not a very good example for application-level checkpointing, but shows that it is never a good idea to rely on obscure third party functions that might be removed. Therefore, I deduct the requirement to only use dependencies in my snapshotting framework, that can be maintained within the framework itself.

EDA applications used for RTL simulations have checkpointing abilities for RTL simulations and some also use some sort of whole process checkpointing for non-RTL simulations, but these are neither portable nor editable. Filling this gap with a snapshotting framework for SystemC/TLM simulations that creates portable checkpoints that are even human modifiable is my goal in this thesis.

Oftentimes, the motivation for using application-level checkpointing is a desire to keep checkpointing size and overhead miniscule. This is also true for the authors of [73]. They worked with a framework that already had checkpointing support, which was implemented in a suboptimal way. The authors showed the importance of the checkpointing process being transparent to the application user, which I take up as a requirement for my snapshotting solution. Their solution displayed very high variance in the overhead depending on the benchmark executed during checkpointing, which seems a bit strange. Therefore, I will not consider this solution for further investigation.

The gem5 simulator presented in [80] is to my knowledge only used in research and not in any industrial applications. It provided a very controlled environment; all models are derived from one base class, not unlike the SystemC kernel in this case. This means to implement checkpointing in gem5, the base class is extended with the checkpointing functions and subsequently the developer have to adapt their models and implement the checkpointing functions. This approach is not really viable for SystmeC as it would break backwards compatibility with existing models and force many developers to extend their models. This might happen in the future when such a change is slowly introduced by the standards committee and there is enough time for everyone to get on board. There are also several proprietary SystemC kernel implementations from tool vendors, that implement the SystemC kernel interfaces and would then also need to adopt the kernel checkpointing interfaces. Another reason why I opted to design my solution in a way that can be easily integrated with either the reference SystemC kernel or a third part implementation.

The work in [13] extended an existing library (dome) for distributed cluster computing with checkpointing functionality. The implementation is not transparent for the user and requires work by the programmer. The authors use preprocessor features to generate code. The dome lib has control over all objects used in the distributed computing tasks in a similar way that the SystemC kernel has control over all objects in the simulation. The authors extended the basic object with the checkpointing functions, which leads to extra work for programmers wanting to use the library, but ensures that each object will be checkpointable. This approach will not work completely for the SystemC kernel, as my goal is to not have to modify the kernel itself, but rather write extensions around it that work with the standardized kernel interfaces. Furthermore, the checkpoints the authors

created are portable which is very important for a distributed system. They were also able to show that not all IO state has to be saved in the checkpoint to be able to restore the state. Unfortunately, the code for the library with their checkpointing extensions is not publicly available. Otherwise it would have been a good candidate for comparison against my own solution. Albeit, with quite some reimplementation effort, porting their extensions to the SystemC kernel or creating something like a SystemC simulation within the dome library. Therefore, I decided to compare my application-level solution against the user-level solution provided through DMTCP.

4.2 Improvements from C++11

C++ as a programming language has several notable defects. Most prominently it has no built-in support for reflection, which greatly complicates the task of serialization. It is not easily possible to iterate over attributes or methods of classes or determine the type of a given object. This downside can be alleviated through external tools like SWIG (as is done in USI) or offline code analysing with Python for example. External tools gain the reflection information by looking into the programs header files.

Furthermore, the underlying type system in C++ is very complicated. When trying to develop a universal serialization or snapshotting library, certain limits are imposed by the typing system. It is not trivial to gain information about a derived class from a virtual base class. In a later Section, I will describe this problem further.

With the introduction of the C++11 standard the language gained several usability enhancements. SystemC does not yet benefit fully from it, but the community is working on it. SystemC requires GCC version 4.8 which already has full support for C++11, so it should not be a problem, to rewrite parts of the SystemC kernel using C++11 coding guidelines. Newer SystemC libraries such as the UVM-SystemC implementation is already written in pure C++11 taking advantage of its new features. The serialization library used in this work is written in pure C++11 as well. With older standards it would not have been possible to have a stand-alone header-only serialization library for C++.

Improvements in C++11 relevant to serialization and checkpointing include:

type inference In older versions of the C++ language it was always necessary to define the type of a variable to use it. C++11 introduces the `auto` keyword, which creates a variable based on the type of its initializer. This is especially useful in templates, where the type of a variable is not always known or does not need to be known. Furthermore, the keyword `decltype` can be used to get the type of an expression that was for example defined using the `auto` keyword.

object construction improvement In older C++ version it was not allowed to expose constructors of base classes to derived classes or call different constructors of the same class in the initializer list of that class. In C++11 these problems do not exist and the resulting code becomes more concise and manageable.

variadic templates In C++11 the number of parameters a template can take is variable, thus enabling the creation of type-safe variadic functions.

variadic macros Improving the compatibility with C variadic macros were added to the C++11 standard.

smart pointers In C++11 there are now three smart pointer types available, `std::unique_ptr`, `std::shared_ptr` and `std::weak_ptr`

With these improvements it is now possible to implement serialization and checkpointing in a more efficient and portable way than before.

4.3 Serialization

In Section 3.1, I mentioned that serialization is commonly used to store the checkpointing information. In the following, I will give a short overview about several serialization libraries with support for C++.

Apache Thrift™ [81] is a software framework for services development supporting many programming languages. The framework is comprised of a software stack and a code generation engine. Thrift™ libraries exist for C++, Java, Python and more languages. The generated code needs to be linked with the Thrift™ library to work. Services can be defined in the thrift language and translated by the code generator into the desired target language. This is not a classic serialization library. By using the services and protocols structure serialization can also be enabled for the supported languages. The C++ library depends on boost, which can be a problematic dependency, especially for cross-platform frameworks. Furthermore, having to regenerate the SystemC code is not feasible. This framework is clearly only an option for newly started projects that can work with a services structure.

Google's protocol buffers (protobuf) are "a language-neutral, platform-neutral, extensible way of serializing structured data for use in communications protocols, data storage and more" [82]. Similar to thrift, protopuf relies on the data structures being defined in advance, which is only natural for communication protocols. This approach is difficult to implement with SystemC models, especially when there is a large library of pre-existing models. Although, it could be possible to serialize TLM transactions this way. Fortunately, protobuf does not require external libraries. Integrating its serialization methodology into a SystemC framework seems too complicated.

The boost libraries [83] are a large collection of C++ libraries that enhance the language by many useful features. Some of these features, like, shared pointers, have made it into newer C++ standards. Boost.Serialization is not yet part of the C++ standard. The idea behind boost.serialization was to be able to reversibly deconstruct C++ data structures. This is a more general specification of the term serialization and the boost library is also quite general in scope. Serialized structures can be transformed to and from a sequence of bytes or textual representation (JSON, XML). Boost.serialization requires modifications on all C++ classes that should be serialized. According to Màrius Montón [84, p. 84], this makes it unsuitable for the use of SystemC serialization, as it would require to rewrite the SystemC kernel to serialize the whole simulation state. In my opinion, the library has a very good approach, but having boost as dependency is suboptimal, for reasons I have stated before already.

MessagePack (msgpack) [85] is another serialization format, similar to JSON. It is supposed to be more efficient, more compact and faster than JSON. It supports many programming languages. The positive is, that with recent C++ version msgpack can be used as header-only library and just be included in a project. The negative is, that it just replaces the serialization exchange format and does not have support for advanced C++ language features. On its

own it is no suitable candidate for C++ serialization, but it could be used to replace the data exchange format of another library to speed up serialization.

Cereal is a pure C++11 library for serialization and was created at the University of Southern California's iLab [86]. It follows the same goals as boost.serialization making it a general use serialization library. It also has a similar feature set to boost.serialization although minus the object tracking code, which make Cereal much faster. Another plus is the usability and good documentation. Cereal furthermore offers facilities for extensibility. The library does not need any external libraries and can be used header-only, which makes it very easy to integrate in existing SystemC frameworks. Furthermore, it is supported by all major C++ compilers, making it also platform independent. Since the library itself is extensible, support for SystemC data types can be added to the library, so there is no need to modify the SystemC simulation kernel. The boost serialization library does not offer this extensibility through simple header files. Since the state of the SystemC kernel is implemented in one specific data type, this data type could be made serializable in theory by simply writing a Cereal extension for that data type.

Apache Avro™ [87] is another framework mainly target remote procedure calls. In its documentation, it is compared to thrift and protobuf. The main difference being no need for code generation. Avro™ uses schemas written in JSON to define data structures. During development the C++ API information for Avro was not accessible, so I can't say anything about possible drawbacks there. Since it is very similar to protobuf and thrift though we can assume, that it is similarly unusable for use in a SystemC simulation framework.

4.4 Continuous Integration for Virtual Platforms

In the previous Chapter, we have learned what CI and UVM are. We have also learned that testing is an integral part of CI. Now we shall have a look at how embedded systems have benefited already from CI. Furthermore, we will look into integration of UVM in testing frameworks with a focus on software development.

Several of the key principles of CI would be difficult to implement with a traditional embedded system development framework. Automating the build-process can usually be achieved with TCL scripts. The same goes for the execution of self-tests, although we can already encounter a bottleneck here. If only RTL models are available the self-tests will take a lot of time and that defies the purpose of CI. The most difficult part would be to implement the testing in the target environment. The target environment of an embedded system would need to be replicated in a simulation system. The environment certainly has some analogue components that are difficult to model in a RTL simulation.

Lwakatare et al. wanted to know why methodologies such as CI and other DevOps[1] practices are not yet adopted in the embedded systems domain [88]. They conducted a case study with four Finnish companies working in the embedded systems domain for several industries. Their methodology involved interviewing multiple employees from each company holding different positions. The authors present four key challenges in the adoption of DevOps practices: (1) Embedded systems software exhibits a strong dependency on the underlying hardware. Often times hardware and software are customized for each customer which makes automation of the release and update process quite difficult. (2) The

[1] DevOps is a combination of the terms "development" and "operations". It comprises technical and non-technical practices for software development.

target environment at the customer site is not visible to the developers designing the test environment. That means that acceptance tests have to be run in the real environment instead of a virtual one which would greatly speed up the process. (3) Technology to reliably deploy updates in the customers environment is not available. Remote updates would interfere with customers acceptance procedures and requirements for high availability of the systems. (4) Usage data is not automatically collected from customers. Monitoring deployed systems remotely by a third party would create security loopholes. Furthermore, Lwakatare et al. identify a lack of suitable tools for embedded-software development as a persistent issue. Although, I have to add that several research groups and some companies are working on this issue, which will be shown later. Unfortunately, the authors do not offer guidelines how to overcome these challenges, instead they refer to the need for more research. At least issues 1 and 2 can be alleviated through using virtual platforms that run in a CI environment with a powerful snapshotting and testing framework.

As mentioned before, key principles of CI are difficult to achieve with traditional embedded system development frameworks. Engblom explores in [39] how virtual platforms can help bring CI to the embedded systems domain. We have seen already how virtual platforms can support testing and debugging. Through scripting APIs virtual platforms are highly automatable. Unlike hardware test set-ups, are always available and configuration changes are possible in seconds. Virtual platforms can also interact with virtual environments to simulate the systems behaviour within an automotive hybrid drive train or a satellite orbiting earth. Depending on the virtual platform the external environment can also be modelled directly within the platform instead of using external interfaces. Since virtual platforms do not have hardware dependencies they lend themselves perfectly to parallel batch processing in a computing cluster or cloud computing environment. Using a virtual platform and integrated simulators alleviates many of the challenges faced when using hardware-based set-ups. Most notably are the testing of fault situations. It is very difficult to inject faults into hardware to exercise fault tolerance and reliability routines [89]. Whereas in simulation faults can be injected into the models with some effort. Furthermore, with more advanced virtual platforms it is possible to dynamically change the configuration of the simulated system during runtime. Functional simulators such as Simics have this ability already, for SystemC based virtual platforms early implementations exist [30, 90].

In the next Section, I will give a short overview of current simulation frameworks for virtual platforms. In the Sections after that, I will look into testing methodologies and the issues of simulating environments and testing for faults.

4.4.1 Simulation Frameworks

The number one challenge for establishing CI workflows for Embedded systems is that they exhibit a strong dependency on physical hardware for development and testing. This makes embedded system development and especially automated testing unnecessary complicated and expensive. Not only are physical development boards expensive and take up space. Oftentimes, they are simply not (yet) available during the development phase. To alleviate these circumstances many people rely on virtual platforms for their software development and increasingly also for developing the actual hardware.

One company developing tools for embedded software and hardware development is Virtutech (now Windriver/Intel). In [91] Magnusson describes Virtutechs virtual testlab approach. He correctly identified testing of complex systems together with software in a

real-world environment as a main challenge. His approach is to use full system simulation for testing with virtual hardware as early as possible and throughout the whole development process. Having virtual hardware offers the advantage to be able to work with the same binary code as on the real hardware while at the same time having as many virtual systems as needed. The amount of virtual testing hardware is only limited by the available computing resources. Having many virtual test systems also allows for interconnection and integration tests that are not feasible with real hardware at a large scale. Virtualised hardware also offers unique insights during the debugging of embedded software. With a sufficiently advanced virtual platform such as Windriver Simics [92] it should be possible to attach standard software-debuggers to the virtualised system and debug the software directly while also having access to hardware debugging information. Simics uses functional abstraction when focussing on software development to get the most performance.

Simics later gained support for checkpointing the state of a virtual platform. Engblom is using the checkpointing feature for the purpose of bug transportation [93]. He argues that transporting bugs with full system checkpoints is much more productive than describing how to reproduce a bug in a bug tracking system. By having access to the same checkpoints a bug reporter and reporter work with the same system state. Having access to a virtual platform with the exact state where the bug occurred eliminates the difficulty of reproducing it on test equipment and furthermore eases the investigation. The checkpoints need to be portable to be transported from one machine to another. They should not be too large and should not contain any machine specific information. Most commercial EDA tools use process-level checkpointing which is not suitable for transportation. Engblom describes several practical scenarios for checkpointing usage which I will not cover here. He mentions the requirement to be able to reproduce inputs that caused the bug as well. These can include internal (Interrupts), external (Network Interface) or spontaneous (Console) inputs. The virtual platform needs to be able to replicate these inputs using scripts and maybe a fault injection mechanism.

4.4.2 Simulating Environments

From the previous Section we have learned, that virtual platforms should be able to replicate inputs to an embedded system. Usually inputs to an embedded system come via attached peripheral devices or sensors monitoring the environment. Environmental influences can also cause faults or unexpected behaviour in embedded system. It would be beneficial to simulate these influences before finalizing the hardware design.

Changes in the environment can be modelled more freely than with a hardware setup. In a virtual environment it is easier to create unlikely events or even use fuzzing to create unexpected events. When a virtual platform supports checkpointing, the simulation state each test can start from a known good state, e.g. after the OS is booted.

Simics is not only used by Swedish researchers and Intel. In [94] Yu et al. present with SimTester an automated interrupt fault testing framework for virtual platforms implemented using Simics. Their focus is on the interrupt system because it can often lead to data race conditions in embedded software. Race conditions are very difficult to test, therefore the authors suggest implementing mechanism that allow observation of all simultaneous transmissions as well as ability to inject transmissions at specific locations. Existing techniques to analyse and detect race conditions require modification of the software code and therefore alter the timing of interrupts. By moving the testing code into the virtual

platform, the software under test can stay unmodified and exhibit its true timing behaviour. Simics already provides APIs for observation and controlling of virtual platform internals. Yu et al. accessed these APIs using Python and implemented the whole framework as a set of Python scripts. To evaluate their system they used real-world examples of deadlocks and race faults discovered in embedded Linux kernel versions. They were able to detect faults that would not have been visible with approaches solely relying on output observation. Their findings led me to select the interrupt controller of the LEON3 SoC as example DUT for the integration with a UVM framework with snapshotting support.

Wenninger et al. used the TLM modelling approach to simulate a network of wireless sensor nodes [95]. They model the environment and communication between the nodes as TLM transactions. To model the nodes a proprietary framework based on a modified SystemC kernel was used. This allowed the authors to model node behaviour using discreet events decoupled from the SystemC kernel timescale. One could imagine that this setup would also benefit from checkpointing and fault injection mechanisms, albeit the authors have not explored theses areas yet.

4.4.3 Testing and Verification Methodologies

Hardware and software components representing embedded systems are often designed synchronously using a co-design methodology. Both hardware and software designs grow in complexity and size, which makes it more difficult to accurately test their respective reliability characteristics, e.g. fault tolerance [96]. Software testing has been addressed in the previous Section. In the following we will look at common hardware testing methods.

At the DATE 2014 conference a panel consisting of leading industry experts and members of academia discussed the verification of SoCs using UVM [97]. The discussion revolved around the questions of how complex SoC verification should be formalized and if it is feasible to extend UVM with SystemC/TLM capabilities or if dedicated tools have to be developed. Verification has to be reusable across multiple levels of abstraction. This means that different teams need access to similar verification methodologies with support for their respective environments and abstraction levels. The reuse of unit-level test benches might not be sufficient to cover complex interactions within a SoC. Reuseability can be achieved through formalization with the help of standards such as UVM and IP-XACT [98]. As of April 2017 UVM is an official IEEE standard [41, 40]. Intellectual Property (IP) vendors have to supply verification IP for multiple abstraction layers. Early system-level verification requires test stimuli to be available. These could be generated and stored in databases. For complex SoCs databases from the big-data domain could be required to store stimuli and verification data. Using software alone for system-level verification might not cover all edge cases and different software exercises a system different. Here, SoC verification can learn from other domains such as server systems and adopt new integration testing methods. Alan Hu argues that “modern SoCs are the most complex devices ever created by humanity”. Furthermore, he suggests that the only scale this complexity is formalization and that the formalized methodologies have to be much easier to use than the non-formal ones. Currently, the interface between hardware and software is specified in plain English which leads to ambiguities and misinterpretations. Formalizing the interface specification would enable automatic test bench and driver generation. Cadence suggests using virtualisation for efficient co-development and co-debug by including the environment the SoC would reside in. Test benches created using TLM for high abstraction-level can be reused for lower

abstraction-levels through attaining to the UVM methodology. This way it is possible to have software as the instrument of verification and reuse and refine test benches through the different verification stages up to actual silicon.

Barnasconi et al. combined UVM with SystemC and its Analog Mixed Signal (AMS) extensions to framework for automotive use cases [99]. Their combination allows the easy transfer from a software based verification of a digital-analogue system onto hardware assisted validation. UVM is extended in three ways: 1. UVM-SystemC-AMS extends the UVM-SystemC API to support simulation of AMS models. For example the driver/monitor UVM components are extended to handle AMS signals, and correctly read/report them from/to the sequencer/scoreboard. 2. Continuous Distribution Functions (CDF) are used in UVM-SystemC-AMS for random input value generation. These input values are generated based on constraints supplied by the test case. 3. Coverage calculation in UVM-SystemC-AMS is done by splitting up the coverage interval into several smaller intervals. The coverage component records values which fall into the smaller intervals and estimates coverage based on the intervals hit. In UVM test bench creation is needed for every individual test. While UVM provides mechanisms to recreate test bench components, there are no mechanisms to for example transfer connections of the DUTs between software verification and hardware validation. A solution to this is the IP-XACT standard [98] and further extensions such as a database which contains test bench configuration and functions to identify and modify individual UVM components. The authors already implemented several suggestions from the discussion described above. Two case studies are presented, one describes a validation test with AMS simulated components, mainly an H-Bridge and the controlling application specific integrated circuit (ASIC) and the other a verification test of an airbag SoC based on an Xilinx Zynq Field Programmable Gate Array (FPGA).

With the standardization of UVM in SystemC already under public review and new standardization concepts for CCI underway, Barnasconi saw the opportunity to build a coherent eco-system around SystemC for system-level design and verification [100]. His previous work already integrated UVM into the SystemC framework at proofed its viability. Both UVM and SystemC use TLM for communication between components. Since both standards were initially developed separately they use slightly different flavours of TLM. Barnasconi explains these differences in detail and offers solutions how to overcome incompatibilities by using new CCI features in SystemC. His proposed solutions are meant to help in discussing the evolution and alignment of the various standards. He identifies the lack of a standardized register API in SystemC as a big integration challenge. In UVM “front door” as well as “back door” access to the registers is required to support fault injection during testing. Solutions such as `sr_register` described in Section 6.1.1 might be a step in the right direction. Although, a solution where the register specification is created in IP-XACT as golden reference is preferred. From this IP-XACT description, register definitions following UVM or SystemC modelling guidelines can be generated automatically.

Since virtual platforms usually run the same binary code as the final hardware, periodic testing can be run with the virtual platform and when the release is near final testing can be done on the real hardware. For the final testing it helps, when the virtual platform provides a way to transfer a virtual platform setup to an FPGA [28].

4.4.4 Simulating Faults

Fault injection methods were already mentioned a few times as necessary features of testing frameworks. Within the UVM-SystemC context it depends on the used virtual platform if this feature is available. The CCI working group has fault injection on their agenda although no proof-of-concept implementation exists yet. A possible implementation could use a scripting interface as described in Section 3.3.1 to facilitate fault injection through the scripting API.

Before SystemC and TLM were established Aidemark et al. developed a stand-alone generic object-oriented fault injection tool [101]. Their goal was to create a user-friendly system that can be adapted to new target systems and fault injection techniques easily. They mainly support two fault injection techniques: 1. Pre-runtime software implemented fault injection, which allows injecting fault into the program and data memory areas before executing the software. 2. Scan-chain implemented fault injection uses built-in test-logic of VLSI circuits to inject faults into pins and some elements representing internal states. These capabilities are very similar to what can be achieved within out UVM framework with the help of checkpoints for internal states and the scripting interface to modify memory areas. Aidemark et al. test their fault injection system with a microprocessor designed for space applications. It is not quite clear from the paper at what abstraction-level they simulate the microprocessor during the fault injection campaigns. Their fault injection tool is written in Java and uses a SQL database to store fault and analytical data. Through templating it is possible to adapt the system to a new target simply by implementing two class functions for the new target. The scan-chain based fault injection technique requires breakpoints to be set where fault should be injected and pausing the execution of the system to do the actual fault injection. A tighter integration of fault-injection mechanism and simulation framework would greatly enhance the efficiency of running fault injection campaigns.

Especially aerospace and automotive microprocessors can benefit from extended testing and verification campaigns using fault injection.

A first step towards fault injection can be establishing checkpointing functionality for a virtual platform. With checkpointing it is necessary to access internal states of the simulation models. It is the accessing of these internal states as well as their modification which will open the door for fault injection in virtual platforms. In the following Section, we will look into the current state of SystemC checkpointing first.

4.5 SystemC Checkpointing

In the previous Sections, checkpointing for virtual platforms was established as a solid foundation for establishing CI on top of it. Most virtual platforms today are written using SystemC, so it seems appropriate to look into the state of checkpointing for SystemC/TLM simulation frameworks more closely. We have also established that fault injection changes the state of simulation models and will also profit from the same techniques employed during the development of a checkpointing feature for a SystemC simulation framework. Checkpointing is furthermore needed for faster tests during CI cycles. The continuous part of continuous integration requires fast build times as well as quick test execution times. Lightening the burden of writing tests can be achieved through a solid testing framework such as UVM, which has been introduced in previous Sections.

Of course checkpointing in the EDA world is not entirely new. George Frazier writes in

his Cadence blog:

> Save and restore (or restart) has existed in HDL simulators for years, but things are trickier if SystemC is involved. For one thing, SystemC simulators use external tools for compilation and linking: i.e. gcc. They have more or less a "black box" understanding of global variables, local variables, file descriptors and heap values that make up the simulation state at any point in time. When you throw in multiple threads implemented with application-level threading packages and the fact that C++ heap objects are impractical to save programmatically, it's easy to see why save and restore tools for HDL simulators can't be easily extended for SystemC. [102]

In the software domain, introspection and reflection are established techniques to peek into the "black box" that is compiled code. Some languages already come with support for introspection and reflection, unfortunately C++, upon which SystemC is based, is not one of them. So researchers and developers try to implement their own solutions to achieve this functionality.

Klingauf and Geffken describe their introspection and reflection framework in [103]. Their framework provides a set of C++ classes that extend the standard OSCI SystemC simulator with reflection, transaction recording and runtime introspection. As they provide a set of classes, only the top-level module requires minimal code modification for the features to be available. Once in place, their functions allow visualization of model hierarchy as UML or XML. Through extensions to the `sc_port` class it is possible to record transactions and stream them to a file for later analysis or to an external tool for immediate analysis and visualization. The described framework can be seen as a technical predecessor to USI presented in Section 3.3.1. The process of discovering the model hierarchy is technically the same in both frameworks. USI offers more flexibility regarding analysis and introspection. Reflection and introspection functionality is an important prerequisite for checkpointing and restore functionality that goes beyond simple process-level checkpointing.

Kraemer et al. (RWTH Aachen and CoWare Inc.) describe a process-level checkpointing implementation for SystemC based virtual platforms in [104]. They specify possible use cases for checkpointing first: Time saving, periodic checkpointing, jump around in time and simulation transfer. Although, they don't go into detail about the implementation of the process-level checkpointing, they mention that it is based on [105], which resulted in the development of BLCR. Supporting simulation transfer with process-level checkpointing is very questionable. For complex systems the resulting checkpoint image will be rather large and process-level checkpointing is usually dependant on the machine the original checkpoint was created on. They integrated the checkpointing features into CoWare Virtual Platform. Drawing from their use case of the Linux boot process on a virtualised PDA platform they formulate general requirements for SytemC checkpoint/restart frameworks: 1. Reliability 2. Transparency 3. Performance 4. Support for external applications 5. Support for OS resources The presented checkpoint restore framework uses process-level checkpointing with some enhancements to support OS resources and external tools. Users need to integrate a special observer object into their modules when they want to be able to save and restore the state. Instantiating the observer and manually calling the necessary procedures to save and restore the module state is not trivial. In my opinion, this does not count as minimal modification. Regarding portability of the resulting checkpoint images,

the authors state themselves that the images are sensitive to recompilation of the virtual platform and require the OS to stay at the same patch level. The checkpoint and restore times are reasonably fast ranging from two to 30 seconds depending on the virtual-platform size. The checkpoint images are quite large ranging from 259 to 1720 megabytes. The size is largely due to the used process-level checkpointing which stores the whole process memory including unnecessary information. We have seen several techniques already which could help reduce checkpoint size in earlier Sections.

Previous works have been done by the two big EDA players Synopsys (2011) and Cadence (2007). Synopsys bought Coware and got their checkpointing solution from them. As explained in [104], Coware developed a process-level checkpointing mechanism for SystemC virtual platforms in 2009. So they could have known about how Cadence is doing it and decided to do it in a very similar way. In Section 3.1, I have already mentioned that process-level checkpointing has the big disadvantages of very large checkpoint size and no portability. Furthermore, checkpoints cannot be altered or tampered with before restore. This would be a useful feature during validation or debugging. One could directly manipulate some internal state in the checkpoint file, without having to change code and recompile.

Synopsys also holds patents related to checkpointing. In the patent text, they even specifically mention SystemC. It seems like they bought the checkpointing functionality together with Coware. The solution described in the patent [106] looks a lot like the work described in [104]. The two authors of that paper affiliated with CoWare are also listed as inventors in the patent. Since they do process-level checkpointing there are also some similarities to DMTCP, which has been described in detail in Section 4.1.3. As one specific example they list GNU Emacs unexec functionality as example for checkpointing of a compiled language program. As this feature relied on now unsupported library features it can be considered as a questionable choice for example. The patent text further describes that user SystemC modules need to be modified to adhere to the checkpoint procedure and be able to communicate with the checkpoint coordinator process. There is no concrete information about the storage format of the checkpoint images so it can not be deducted if the checkpoint images are portable. Although, since they do process-level checkpointing it can be assumed that the portability is very limited.

Cadence checkpointing solution called *save and restore* within the Incisive software suite is limited to Linux platforms. This hints towards their implementation of checkpointing. The blog post by George Frazier explains a few details of their checkpointing process. Apparently, they use a full memory dump to save and restore the simulation state. This makes the checkpointing feature quite limited. A developer will only be able to restore the checkpoint on the same machine with the same libraries and the same model version. It could very well be possible that if after saving the simulation state an automatic system update install new libraries or even a new kernel version, restoration of the saved state will not be possible any more. Although, I have to admit that cadence deserves some credit for this solution. It requires quite some effort on the simulator side and it was implemented four years earlier than the Synopsys solution.

Engblom has analysed the save and restore functionality of Cadence [107]. Apparently, Cadence has implemented process-level checkpointing and simply creates a memory dump as checkpoint. They do not even support handling of file descriptors, the user has to implement functions if he wants to restore a model that uses files or other OS resources [108].

As with other process-level checkpointing implementation the functionality is limited to the same machine and same code revision of the virtual platform. Its usefulness is therefore limited to saving start-up time [102] for one developer. Advanced checkpointing use cases are not supported with the Cadence tools.

In 2009 GreenSoCs and Virtutech were working with the university of Barcelona on another approach to SystemC checkpointing [84]. Virtutech, which is now called Windriver, has developed Simics, a full system simulator that has already been mentioned and evaluated in Section 4.4.1. They use the existing application-level checkpointing functionality of Simics and created simulation adapters that allowed them to simulate SystemC models within the Simics simulator.

The main work lies in the simulation bridge between Simics and the SystemC kernel, which handles translation of transactions and synchronization. The same methodology has been used to integrate QEMU and SystemC, both with QEMU has master and with the SystemC kernel as master. By integration into Simics Virtual Platform SystemC looses its appeal of an open platform. Simics is neither open nor free to use and therefore not an option for most researchers exploring system-level design methods. In Simics the integrated SystemC simulation operates as slave and is controlled by Simics. Through the use of Greencontrol simulation and model parameters which use the `gs_params` decorator are made available as Simics attributes. This way they can be accessed by Simics application-level checkpointing functionality.

Full SystemC checkpointing ability was only achieved by also modifying the SystemC kernel. The event queue code was extended with a function to read out the event queues without modifying them. This could also have been achieved through C++ language features without the need to modify the source code as I will demonstrate later. Furthermore, the kernel was extended with a checkpoint manager class that handles all checkpointing related tasks.

Model checkpointing is not implemented in a transparent way. They still had to adapt the SystemC models and not all SystemC features where supported. Most notably model developers are not allowed to use `SC_THREAD`s with this setup. Variables that are wrapped in GreenSocs parameters are automatically saved and restored. The user also has the option to implement explicit save and restore function for their models.

Monton extended the primitive channel classes included in the SystemC kernel with save and restore functionality. It is unclear if the modifications he has done for version 2.2.0 of SystemC can easily be ported to a newer SystemC version. As I have stated earlier, current SystemC version have deprecated several functions related to `sc_simcontext`. Relying on a proprietary checkpointing solution while also needing to modify the SystemC kernel is a double loss. Not being able to use `SC_THREAD` might even be a good feature and enforce event driven model design.

Monton states in his related work that including a serialization library in SystemC would need too much kernel modifications. He did not mention that this way the checkpointing would be more transparent to the user and available in stand-alone simulations. His version still needed kernel modifications and only works within a proprietary platform. On the other hand, the snapshot files are relatively small and portable. Checkpointing overhead is also quite small with only 2%, although they don't mention the performance penalty from running SystemC within another simulator.

In [109] Engblom explains why threads are almost impossible to checkpoint in SystemC simulations. The threads are managed by an external OS library which contains inaccessible state. The thread stack and program counter vary between each code version and therefore cannot be reliably restored from a previous state. There are solutions to these problems which have been explained in a previous Section, but they come with the downsides of process-level checkpointing. Advanced C++11 language feature and modern serialization libraries might help achieve what Engblom thought almost impossible in 2009.

In 2009 the CCI working group of the Open SystemC Initiative released the first version of the requirements specification for configuration interfaces [110]. Their goal is to have a common interface for configuration parameters used in SystemC virtual platforms. The parameter implementation shall be tool agnostic and portable. The parameters follow the name value pair concept and are stored via JSON. This way the parameters can be stored in a human readable and modifiable way as well as implemented in many tools. As future plans the working group also lists the implementation of save/restore functionality on top of the configuration parameters. As of 2013 a draft implementation [111] for CCI parameters exists and as of 2016 an updated reference implementation based on works by GreenSocs and Ericsson is available [112]. The SoCRocket framework come with it's own parameter implementation, *sr_param*, which is compatible to the CCI specification. In Section 5.7, I will evaluate if and how these parameters could be used for rudimentary application-level checkpointing.

While Monton was working on his thesis, Carbon Design Systems was working together with ARM on their fast models with checkpointing support. Neifert describes in [113] how Carbon Designs "Swap n Play" uses checkpointing to enable combined fast and accurate simulations. While the simulation runs with the ARM fast models it takes periodic checkpoints of the system and continues running. When the fast simulation is done, the created checkpoints can be used to start up accurate simulations. Since many checkpoints were created, several accurate simulations can be run in parallel on a distributed computing system. In [114] the checkpointing support in the ARM models is explained in more detail. The fast as well as the accurate models make use of ARMs ESL APIs. The cycle accurate ARM models are automatically created from the RTL sources. Since both model variants use the same underlying APIs it is possible to create checkpoints with the fast models and load the state into the accurate models within some limits. Register-side-effects need to be considered. Furthermore, caches or pipelines are not saved and neither restored. If a developer plans using other models than the ones provided by ARM, the save and restore functions from ARMs cycle accurate debug interface have to be implemented. Neifert further explains the testing process needed to verify the checkpointing functionality. Their approach is to create many random checkpoints and then restart simulation from these random checkpoints to see if the simulation can still finish.

Using the ARM APIs for checkpointing is an interesting approach, but ARM models and APIs are not generally available. A standardized approach which is compatible or comes with the SystemC/TLM reference implementation would be preferable. Neifert also does not give any details regarding portability of the created checkpoints. It has to be assumed that their are slightly portable, since he mentions the possibility to start checkpoints on multiple machines in parallel. It can be assumed that using Carbon Designs and ARMs "Swap n Play" feature is not compatible with other register implementations. According

to the ARM ESL API documentation [115] the save restore functions work automatically with the ARM register and memory interfaces. For custom models saving and restoring of simple variables is supported through a data stream similar to standard input output streams. The developer has to do manual serialization here, when he uses complex data types. From these facts I assume that Carbon Design implemented their checkpointing solution at the application level.

Xu investigated previous SystemC checkpointing works by Monton and Kraemer, but was not satisfied with them. His approach extends the SystemC kernel with checkpointing functionality [116]. The checkpointing itself is done by the MTCP library described in Section 4.1.3 above. The checkpointing functions are added as member functions to `sc_simcontext`. Furthermore a new simulation phase *checkpoint notification phase* is added with its own checkpoint event queue. This way it is possible to queue checkpoint times at certain points in simulation time. With a MTCP extension it is also possible to queue periodic checkpoints at real time. Since the SystemC kernel synchronizes its threads at every delta step, the MTCP library did not need extra thread support. Restoring a saved simulation checkpoint requires an external tool that reconstructs the memory of the original SystemC platform executable. Xu modified the MTCP library to be able to save multiple checkpoints as by default it was overwriting the last checkpoint. Furthermore, he fixed a bug in MTCPs memory reallocation code which led to constant segmentation faults during restore. During performance measurements with a simulated shift register he measured checkpoint time of less than half a second and checkpoint image size less than 2 megabytes. Unfortunately, no experiments with more complex systems were done. The portability of the checkpoint images was not tested here, but has been proven by the DMTCP [65] developers. The approach is interesting, since it does not require modifications of existing models or puts restrictions on the developer. The kernel modification would need to be approved for inclusion in the reference implementation, which is unlikely since the `sc_simcontext` function have been marked as deprecated in the latest SystemC release (Version 2.3.1). Furthermore, the reliance on an external library with questionable reliability is suboptimal. Xu already found and fixed one bug in the MTCP library without support from the original developers.

Restoring a previous simulation state is analogous to instantiating simulation models inside a virtual platform from a predefined configuration. Sauer and Loeb describe their infrastructure for dynamic creation and configuration of virtual platform in [90]. They use Cadence's CCI parameter implementation for the configuration part of their infrastructure. The dynamic creation part follows the classic factory design-pattern [117]. The authors state that currently users must deal with long start-up times during pre-silicon software development. Dynamic creation allows creating models with varying parameters during runtime. This approach can be akin to checkpointing if all internal state parameters and variables of a model are exposed as configuration parameters. They have implemented the configuration parameters and model factory as C++ library without the need of SystemC kernel modifications. Model parameters are stored as *boost property tree* [118]. Serialization and deserialization is possible through the use of boost's `lexical_cast` [118]. This allows storing of parameter configurations in JSON. The factory even supports loading pre-compiled models from a shared library. The infrastructure is evaluated with a multi-processor virtual platform used in telecommunications. Using dynamic creation with the factory introduces a 5% overhead compared to static configuration. The speed-up becomes apparent when

top-level and model parameter changes are applied. Here, a speed-up factor of 2 for top-level changes and up to five for model-level changes can be achieved. Using a model factory is a promising approach to circumvent several of the downsides of traditional checkpointing. Although, it requires a lot of care of the model developers as they have to expose all important variables. Furthermore, unlike process or application-level checkpointing, there is no support for external resources. The infrastructure described by Sauer and Loeb could be integrated into a checkpointing framework to overcome shortcomings of both.

Tabacaru et al. present a rather convoluted approach to checkpointing in [119]. They developed a checkpointing mechanism specifically to speed up verification of safety-critical automotive applications. Their starting point are Verilog based RTL simulations. RTL simulation tools already offer checkpointing functionality through tracing, but the authors argue, that these checkpoint files are far to large and not usable for their use-case. Verilator is used to convert the Verilog models into cycle-accurate SystemC models. The checkpointing mechanism is then integrated into the resulting C++ code. Furthermore, they extend the generated code with abilities for fault-injection. A checkpoint image generated with their mechanism contains signal traces from the models in VCD-file format and internal state variables from Verilators cycle accurate simulator stored as comma-separated values. A Python application is used to extract checkpoints from a fault-free reference simulation run at certain simulation times. During verification, the simulation can then be restarted from those checkpoints and faults can be injected. They achieved much better performance with less disk space used compared to commercial RTL simulator checkpointing. Although, with application-level checkpointing of the generated SystemC models and integration into a UVM framework that supports fault-injection they could have achieved much better results.

4.6 Summary

The various checkpointing methodologies and their implementations were already summarized in the respective Sections above. With respect to the integration into an efficient CI workflow only user-level or application-level checkpointing seem feasible solutions. System-level checkpointing is too low-level and its main focus is ensuring reliable operation of systems. Virtual-machine-level checkpointing surely is the most prevalent method right now, but it does elude the requirements of small and modifiable checkpoints.

User-level checkpointing requires support from external libraries as well as some modification to application code. Application-level checkpointing requires that the application be written in a language with advanced features such as introspection and reflection as well as some extra language modules or libraries. With the introduction of the C++11 standard, C++ gained several features that make implementing application-level checkpointing less of a hassle. It is possible now to hide most of the adaptations from the user and thereby make the checkpointing process transparent.

The various works evaluated in the above Sections show that serialization is the best way to store checkpoint data. Not many languages come with built-in serialization support. There are several interesting libraries for C++ that implement serialization, which will be explored in Section 5.1.

Establishing CI for embedded system design and development is only possible through the use of virtual platforms to simulate embedded systems and their environments. This in turn means introducing a powerful test and verification framework such as UVM to

cover the testing aspects of CI. Another important part of CI is very short build and test execution times to allow for fast turnaround times when changes are detected in the code base. Test execution times can be reduced immensely by introducing checkpointing in the test environment, so that the desired state that is needed for the test execution is reached almost immediately without spending much time in simulation. As most virtual platform frameworks are implemented with SystemC, checkpointing has to be implemented inside the SystemC simulation framework. Since there are multiple implementations of the SystemC simulation kernel available, the checkpointing implementation should work with all of them.

Checkpointing together with reporting and logging facilities provided by virtual platforms provide much more detailed feedback to developers than hardware test set-ups ever could.

Kraemer et al. created in [104] an early SystemC checkpointing implementation at the user level. They used the "observer" design pattern to augment simulation models for checkpointing. The checkpoints resulting from this is implementation are not portable and rather large. Although, some modifications might be necessary they should be kept minimal. The checkpointing functionality should not interfere with those. From the works reviewed in the previous Sections, I compiled a list of SystemC checkpointing requirements:

R1 Reliability

The restored simulation should be exactly the same state as the original.

R2 Transparency

The checkpointing should be transparent to the user and not require modifications to the model code. Although, some modifications might be necessary, they should be kept minimal.

R3 Portability

The checkpoints should be independent of the OS libraries and Kernel versions. A user should be able to create a checkpoint on one machine and restore it on another.

R4 Modifiability

The checkpoint format should support manual modification by users. This can either be through a helper application or through direct modification in an editor.

R5 Performance

The checkpoint and restart procedures should not have significant impact on the simulation performance. Furthermore, the added complexity in the platform code should not impact recompilation time significantly.

R6 Support for external applications

Virtual platforms are often used in conjunction with other tools like consoles or debuggers. The checkpointing functionality should not interfere with those.

R7 Support for OS resources

The checkpointing functionality has to be able to handle open files and I/O streams as they are frequently used in SystemC virtual platforms.

R8 Self-reliance

The checkpointing functionality should be contained within the specific simulation framework or even the SystemC simulation kernel. It should not rely on third party software. Reliance on external libraries is okay, as long as they are bundled with the simulator.

These requirements still hold true. They are very useful for evaluating the other SystemC checkpointing implementations as well as my own.

There is already rudimentary checkpointing functionality available in popular EDA tools. Synopsys holds a patent [106] on a user-level checkpointing mechanism that seems very familiar to DMTCP. In Chapter 7, I will compare DMTPC against my own checkpointing solution. The Synopsys checkpoints are likely not portable nor modifiable due to the nature of user-level checkpointing.

Cadence has only very limited user-level checkpointing [108]. The user has to implement his own functions for saving and restoring file descriptors for example. Furthermore, the checkpoints are not portable or modifiable.

Monton created a well documented [84] application-level checkpointing implementation for SystemC simulations. Albeit, using the SystemC kernel within the Simics simulator. This does not count as plain SystemC checkpointing. Internal states of models are exported through parameters that resemble CCI parameters. The SystemC kernel event-queue handling was adapted and a checkpoint managing class was added to the kernel. As I will demonstrate later, these steps are not necessary when using advanced C++ features. The checkpointing implementation is not transparent. The user has to take care to use parameters extensively and is not allowed to use *SC_THREADS* at all. The user can implement custom save and restore function within his model. This seems to be a common theme for all referenced checkpointing implementations. Actually, this is a good option to allow advanced users to adapt the checkpointing implementation to their exact needs. Monton's implementation relies on now deprecated SystemC kernel functionality, so it is not very future-proof. On the upside, he achieved small and, due to application-level checkpointing, portable checkpoints. Although, this achievement is more attributable to the Simics simulator.

Carbon Design implemented [113] application-level checkpointing for SystemC simulations that use the ARM fast models. They only checkpoint registers and memory. The checkpoint data is written in a simple binary stream with manual serialization implementation. Not many details are available for this implementation. In [114] the authors describe how they use the Carbon Design checkpointing feature for extensive testing of models. During these test sessions they use the checkpoints for fault injection by randomizing the checkpoint data. So it seems the checkpoints are modifiable. There is not data available for their size or if they are portable.

In [116] the author extended the SystemC kernel for checkpointing. Again relying on deprecated functionality like Monton before. The author used the process checkpointing library MTCP for the actual checkpointing which is part of the DMTCP user-level checkpointing implementation. As stated before, these checkpoints are neither modifiable nor portable.

During the restore phase of a checkpointing implementation the "factory" design pattern could be helpful. The authors of [90] explored using the "factory" pattern in conjunction

	requirements							
related work	R1	R2	R3	R4	R5	R6	R7	R8
Kraemer [104]	+	-	-	-	+	+	+	+
Synopsys [106]	+	-	o	-	+	+	o	+
Cadence [102]	+	-	-	-	+	o	-	+
Monton [84]	+	-	+	+	+	o	-	-
Neifert [113]	+	+	+	o	o	o	-	+
Xu [116]	+	+	+	-	+	o	o	-

Table 4.1: Evaluation of related work regarding requirements

with CCI parameters for something that seems like checkpointing, but only covers the start-up phase of the simulation. Models can be instantiated with a specific configuration, but there is no mentions of advancing simulation state or storing the state of a running simulation. Although, in theory it would be possible if all internal state variables are exposed as CCI parameters. This would then be quite similar to Montons work in [84].

The authors of [119] describe their very convoluted way of creating and using checkpoints. Their work shows that there are use cases, apart from integration into a CI workflow, for SystemC checkpointing paired with a powerful verification framework such as USI.

Table 4.1 sums up my findings from related work review. In this Table, I compare each related work that implemented SystemC checkpointing against the requirements I have formulated above. In the right columns of the Table a "+" indicates that the requirement is met. A "-" shows that this requirement is clearly not met. Whereas a "o" shows that it is unclear if this requirement has been met.

The Table quickly shows that there is no solution available that meets all the requirements. Although, there are some such as the solution from Xu or Monton that show the most requirements met, but they both rely on third party software for the actual checkpointing part.

From the existing work I can deduct, that the requirements of Portability and Modifiability of checkpoints can only be achieved when using application-level checkpointing. Transparency can be created with the use of modern C++11 features as well as a matching serialization library. Performance depends on the actual implementation and has to be evaluated later. The same is true for reliability. These requirements will be observed during the design phase already.

The idea of using application-level checkpointing can certainly be recycled. It just needs to be implemented in a way that meets the requirements for transparency and performance. Adding more complex code such as a library for serializing checkpoint data will surely affect performance. Meeting the transparency requirement will be the most challenging at will require use of modern C++11 features as well as ingenuity to hide the complexity from the user and make checkpointing available in a straightforward manner that will not require much intervention in user code. The same goes for implementing support for OS resources. The complexity of the save and restore functions needed to checkpoint these data structures will be abstracted away. Supporting external applications will not break as long as the SystemC kernel stays the same. Modifiability of the checkpoints will be given by the selected output data format of the chosen serialization library.

In the next Chapter, I will describe the architecture of my solution for a SystemC checkpointing implementation that works with the OSCI SystemC kernel implementation and should also work with proprietary implementations.

5 Architecture and Concepts

Thomas Becker writes in his essay about the tension of object-oriented versus generic programming in C++:

> Good engineering involves compromise at every turn. A good, working, finished product is never pure by the standards of any one idiom or methodology. The art of good engineering is not the art of discovering and applying the one right idiom over all others. The art of good engineering is to know what your options are, and then to choose your trade-offs wisely rather than letting others choose them for you. [120]

SystemC is a C++ library for modelling hardware components and systems. It is designed using object oriented paradigms. Good SystemC models follow these object-oriented paradigms to create efficient and elegant models. There are also quite a few SystemC model developers that come from the traditional hardware side and tend to follow more generic programming paradigms. These models then usually work, but are less efficient and not as flexible when it comes to updating the surrounding framework and libraries.

A SystemC simulation framework should also assist developers with modelling guidelines and provide functionality and methodologies for efficient model development. Efficient model development meaning here that a developer should focus on functional aspect of the model and not on how to work with a specific framework.

My goal with this work is to create a self-contained and portable snapshotting solution for SystemC simulations. The solution contains all the necessary code and instructions for a developer to extend their simulation framework with it. Model adaptations will not be necessary.

In Section 4.6, I have already described my requirements for SystemC snapshotting. To recall, the requirements were the following: *R1* Reliability; *R2* Transparency; *R3* Portability; *R4* Modifiability; *R5* Performance; *R6* Support for external applications; *R7* Support for OS resources; *R8* Self-reliance As mentioned before, these requirements lead to the conclusion that snapshotting has to be implemented at application level.

Application-level snapshotting requires a serialization library to support writing and restoring complex data types. There is no built-in serialization with C++, but that does not mean I have to write it from scratch. There is a large number of serialization options available for C++. In the next Section, I will explain my reasoning for going with a certain library.

There is no ready SystemC checkpointing solution that fulfils my above requirements. However, parts from other SystemC extensions can be reused for checkpointing. For example, the introspection mechanism of USI can be used to discover the SystemC model hierarchy. Montón's idea of using checkpointable parameters was good, but not executed well. It can be done without the need for an external simulator that wraps around the SystemC simulation. Limiting the use of parameter types to actual configuration types and still be able to checkpoint the necessary variables to restore a model's state is certainly

feasible as I will show later. Using a snapshot manager class to manage the snapshotting process is also quite straightforward and has been done before. However, the solution presented in [116] implemented the snapshot manager within the SystemC kernel, which breaks portability of the snapshotting solution. The reference SystemC kernel should not be modified. This restriction makes accessing certain data structures difficult, but not impossible, as I will explain in Section 5.3. Ultimately, the resulting code will be more portable and compatible with other SystemC kernel implementations.

The main component of the snapshotting framework is naturally the serialization library. The library is already mostly complete and needs to be integrated in the SystemC simulation. The integration is handled through addition of a snapshot manager class. This class will be derived from the `sc_module` base class and contains all the functionality needed to snapshot SystemC simulations.

In the following Sections, I will describe how I designed the snapshot manager class to meet all the requirements which have been specified in the previous Section. In Section 5.1, I will detail out the differences between several serialization libraries available for C++ and show which one is the best for my use case. Having decided on the serialization library to use, I am ready to show the design of my snapshot manager class in Section 5.2. Important C++11 concepts for accessing class members transparently are described in Section 5.3. These concepts are then applied to enable my snapshot manager class to access and modify the SystemC kernel simulation time in Section 5.4. In Section 5.5i, the extended phase callbacks of the SystemC kernel are used as triggers for our snapshotting process. Section 5.6 shows the USI mechanism that is used to scan the model hierarchy. Some SystemC extensions need to be adapted to be checkpointable. These modifications are explained in Section 5.7. The final Section of this Chapter contains information about how virtual functions complicate the snapshotting process and what can be done about it.

5.1 Serialization Library

Overall the snapshotting framework should not rely on external tools, so it can be easily integrated with various simulators supporting SystemC/TLM standards. Portability is another important requirement. The framework should not rely on platform or OS specific functions.

A big part of snapshotting is organization and storage of the snapshot data. As has been explained in Chapter 4, serialization is a common technique for storing various data structures in a portable fashion.

There are of course a plethora of serialization options when it comes to C++. From the overall requirements for a SystemC snapshotting frameworks, I derive the following requirements for a serialization library:

- The code should be open sourced and available under a compatible license.
- Compilation with GCC (also older versions) should not be problematic.
- The library has to be platform independent.
- The serialization overhead should be small.
- The library should be easily extensible.

- Serializing a smart pointer (or reference) should automatically trigger serialization of the referred object.
- Serialization of complex data models from scalar fields (bool, int, float) to containers (vector, list, etc) should be handled transparently.
- Support for non-intrusive serialization for objects where the header file cannot be modified.
- Serialized data should be available in a human-readable format.

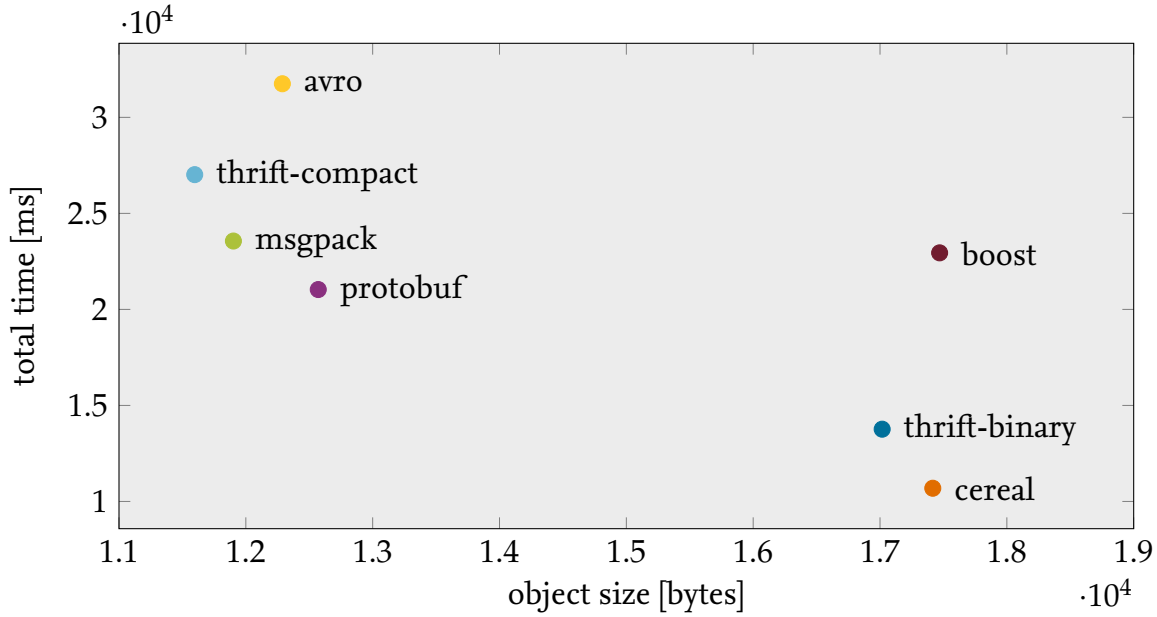

Figure 5.1: Serialization library benchmark results from [121]

Konstantin Sorokin has published benchmarks for the libraries I have listed in Section 4.3 on Github [121]. The results are presented in Figure 5.1. The numbers were generated using a typical desktop CPU (Intel Core i5). Each benchmark run contains 1000000 serialize-deserialize operations and is run 50 times, before averaging the results. These benchmarks help making a final decision for the serialization library. In the Figure, two groups can be made out. In the upper left area are the slow libraries with small object size and towards the right side are the faster libraries with larger object size. Boost stands out by being rather slow and having a large object size. Speed of serialization is much more important than the final size of the serialized data. Storage is cheap and fast nowadays, but during complex system simulations the CPU is preoccupied, so the serialization process should not slow down the simulation significantly. With this in mind, the choice falls easily towards the *Cereal* serialization library. As mentioned before, it has a similar feature set to Boost.Serialization, but it is header only and therefore much more portable. It can simply be included in any simulation framework that supports C++11. Since SystemC requires a fairly new GCC, the build environment has support for the C++11 standard. Although it is already a few years old, it has not permeated SystemC completely yet [122].

5.2 Managing the Snapshotting Process

Snapshotting should be possible to manage from inside the `sc_main` or with external tools hooking into available SystemC APIs. This is achieved through introduction of a snapshot manager class which is derived from a `sc_module`. The snapshot manager module provides data structures for storing SystemC model checkpointing data as well as functions for saving and loading snapshots.

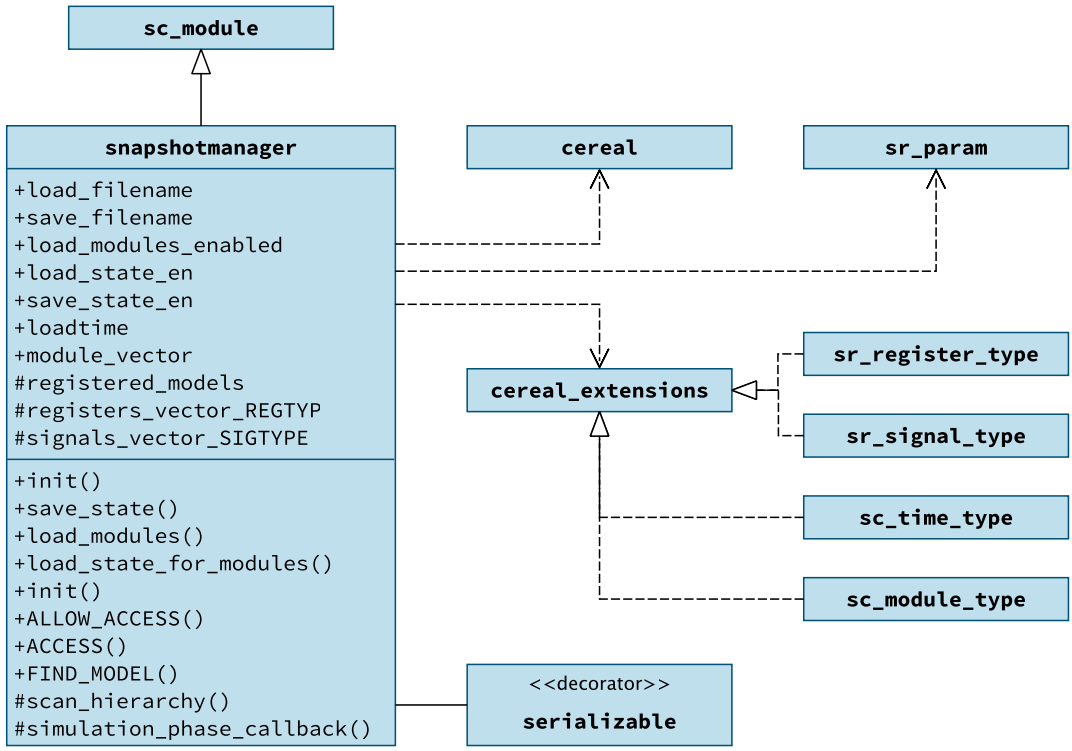

Figure 5.2: Class diagram of the snapshotmanager class

Figure 5.2 shows the relationship between the snapshot manager class and it's associated classes. The snapshot manager class is derived from a `sc_module`, so that it can access SystemC kernel features such as the extended phase callbacks described in Section 5.5. Furthermore, it is possible to access its functionality from USI scripts. This way the snapshotting process is more flexible and accessible to users of the simulation framework. The snapshot manager class uses `sr_param` to expose configuration parameters such as file names of snapshot files. These can then be changed during runtime via USI. The Cereal library is included here together with several extensions for SystemC data types. The implementation of the Cereal library extension will be explained in detail in Section 6.2.4.

The snapshot manager handles loading of save state data through SystemC phase callbacks which will be explained in Section 5.5. Saving the state can be done at any point during simulation by calling the `save_state` function of the snapshot manager class object. The snapshot manager class stores the data types that shall be checkpointed within several vectors. Discovering checkpointable data types is implemented in the `scan_hierarchy()` function and explained in Section 5.6.

In the next Section, I will explain how advanced C++11 features can be used to transparently implement checkpointing even for data types that cannot be modified or accessed.

5.3 Non-intrusive serialization

In Section 4.6, I stated transparency as one requirement for SystemC checkpointing. In the context of serialization, transparency means to be able to serialize objects where the implementation is not modifiable in a non-intrusive way. This requirement is already quite important to access private and protected values inside the SystemC simulation kernel. Modification of the SystemC simulation kernel would limit the snapshotting framework to only work with that specific simulation kernel version. This would lead to incompatibilities with commercial simulation tools and render the whole effort rather pointless. As the requirement is to create a standard compliant framework which can be used as a non-intrusive add-on to existing simulation setups. The requirement also applies when external model libraries are used which might come as binary only.

C++ does not offer built-in facilities for runtime introspection and neither does SystemC, but the macro and templating engines of C++ are very powerful tools and can be used to access private member variables of classes without modifying the class itself. Modifying the class itself should always be the preferred way, as there are reasons for data members being private. Although in some situations that is not possible. For example, when only header files are available and the implementation is hidden.

Schaub described a safe way of accessing private member variables in his blog [123]. A previous incarnation of the technique was already suggested for inclusion in the Boost serialization library. As the Cereal library had no such functionality in place already, the necessary code needs to be added to the snapshot manager class.

In C++, namespaces usually prevent accessing functions in other namespaces if the full name is not explicitly specified. During argument-dependent lookup (ADL), it is possible to widen the namespace search space for unqualified function names, depending on the types of arguments to the function call. A good example for ADL is the standard « operator used with `std::cout`. While `cout` requires the namespace identifier `std` to be found, the overloaded operator « is looked up by the compiler through ADL. The overloaded « operator is not part of the `std` namespace, but through ADL it is possible to let non-member functions seem like they are member of a class.

ADL is one part of the puzzle to get access to otherwise invisible private class members. Other puzzle pieces arrived with the introduction of C++11. The technique described in [123] would also be possible without C++11 features, but they make it much more convenient and safer to use.

Before going into more detail about how to access private class members from outside of the class namespace we need to explain three significant C++11 features.

`decltype` `decltype` was introduced in C++11 to enable deduction of an expression's type at compile time. The expression can be a simple variable or any valid operation involving multiple variables. If the deducted type is a user defined type, it is even possible to access nested types directly or use `decltype` to specify a base class.

`remove_reference` `remove_reference` allows getting the non-reference type of a variable reference. If the variable is not a reference the same type is returned.

`nullptr` `nullptr` represents a universal null pointer. In the past, the macro `NULL` or the literal value 0 was used as a null pointer value. That was often buggy and unsafe as a macro could easily be redefined and not be 0, which would lead to unexpected behaviour. `nullptr` is a literal of type `std::nullptr_t` similar to `true` or `false` which are of type `bool`. The null pointer literal can be used in conditionals, as initial value for variables, as function argument, but not on the left side of any assignment.

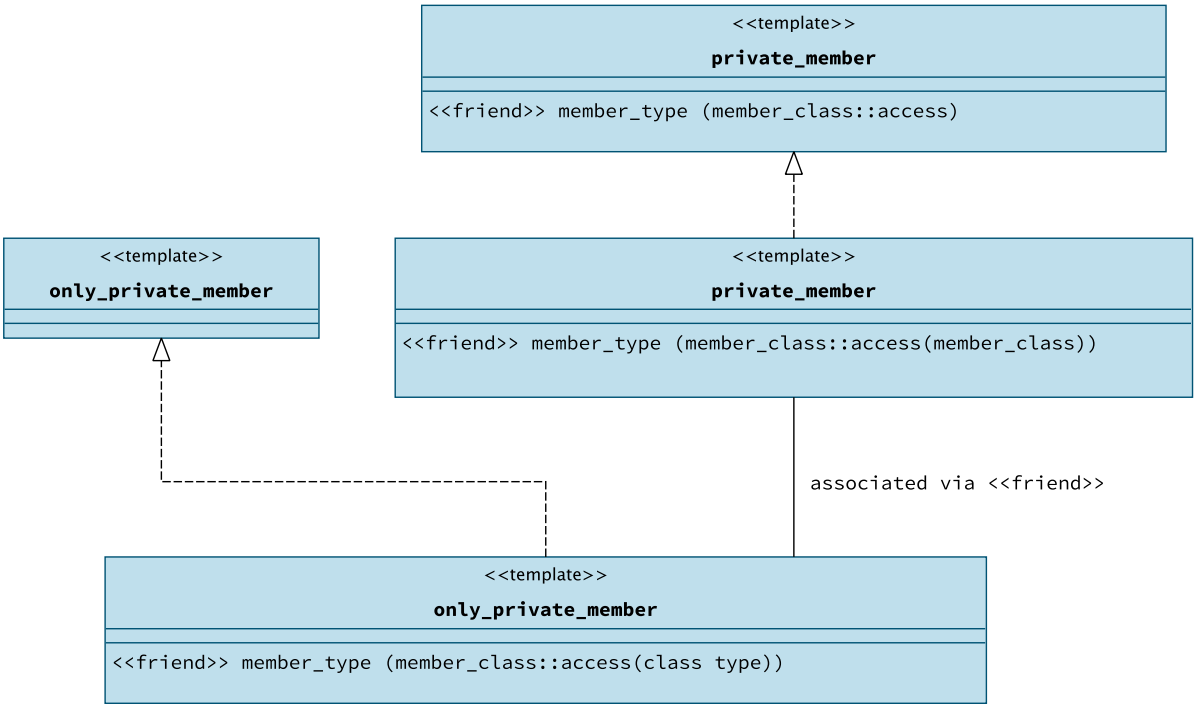

Figure 5.3: Class diagram for private member access templates

The class diagram in Figure 5.3 gives an overview of the template structure needed to access private members of classes. The implementation of the above described concepts can be found in Section 6.2.1. In the next Section, these concepts are used to access the internal time variable of the SystemC kernel.

Another necessary component for transparent checkpointing are the extensions to the Cereal serialization library. Figure 5.2 already shows the relationship of the Cereal extensions to the snapshot manager class. The following paragraphs provide some more details on how these extensions are designed.

The Cereal library makes extensive use of C++11 features. It comes as header-only implementation. Extending Cereal with support for new data type requires teaching the Cereal class how to deal with the new type. The save and load functions have at least two parameters, the archive and the value that should be stored or loaded. Both parameters of course have types. Cereal has support for multiple data types. Using the overloading function technique, many combinations of archive type and value type can be handled by the same function calls. Cereal included already support for many standard C++ data types. These implementations can be taken as example for own implementations. The diagram in Figure 5.4 shows the

relationship of the classes used for extending the Cereal library with support for the `sc_time` data type. For the actual implementation, using templates makes it possible to support many different data types.

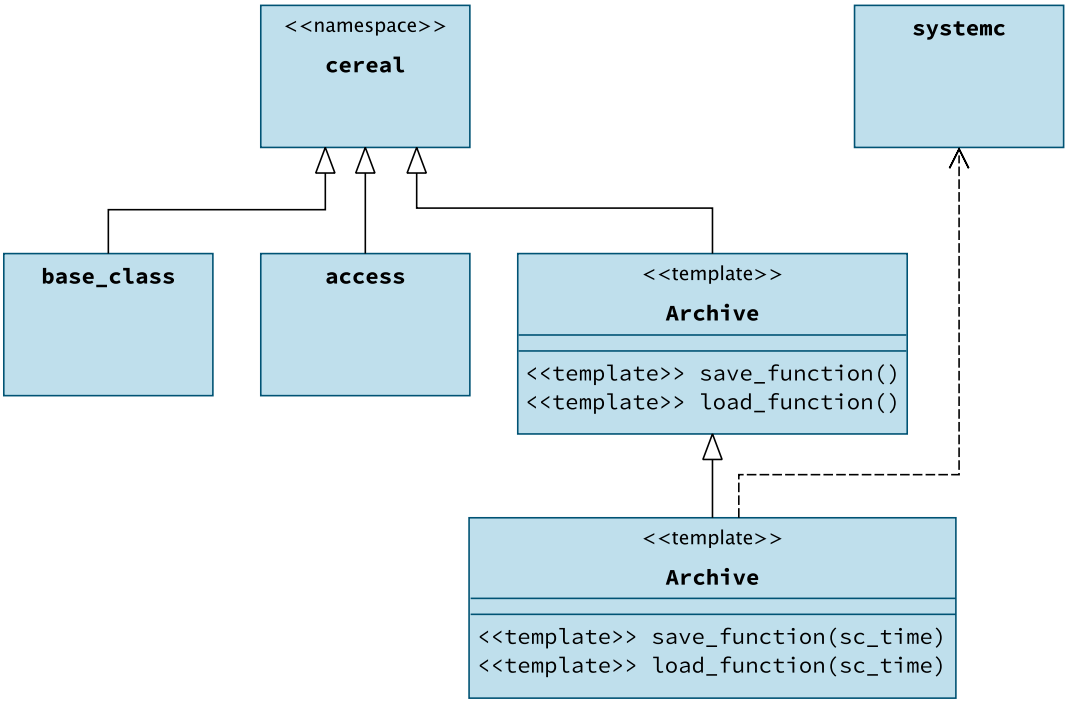

Figure 5.4: Class diagram of a Cereal extension using the example of `sc_time`

5.4 Restoring Simulation Time

The SystemC kernel does not support explicitly setting the simulation time or jumping ahead in time from the start of simulation. Time can only be advanced in one direction and every advancement of time has side-effects in the simulation.

A first idea was using the `tlm_quantumkeeper` class included with the SystemC kernel to modify how time is counted. Unfortunately, this only works on the model level and does not change the global simulation time stamp. The global time stamp is advanced with every delta cycle of the simulator.

Earlier, we have learned how private members of classes can be accessed for snapshotting and restoration. The current simulation time stamp is stored in the `m_curr_time` private member variable of the `sc_simcontext` class. The snapshot manager class contains the `ALLOW_ACCESS` macro call to allow it to access the time stamp stored inside the simulation context. Now it is possible to simply overwrite the simulation time stamp during the load function. The snapshot manager class also has its own public member variable `loadtime` which stores the loaded simulation time stamp so it can be used in the `sc_main` for calls to `sc_start`, so that the simulation continues from the correct time stamp.

With some effort the Cereal library could also be extended with snapshotting functionality for the `sc_simcontext` class. However, this would result in major restructuring of the snapshot manager class and the snapshotting process itself. Although, this would be

the only option if someone wants to restore the full event queue of the SystemC kernel simulation context.

The current snapshot manager relies on the fact, that a call to `sc_pause` precedes the calls to its `save_state` function, thereby ensuring that the event queue is empty. The queue will be refilled upon restarting of the simulation and restoring the previous state of the attached models.

5.5 Extended phase callbacks

The snapshot manager needs to be able to operate only during specific SystemC simulation phases. Otherwise it would break the simulation and snapshot data could be corrupted.

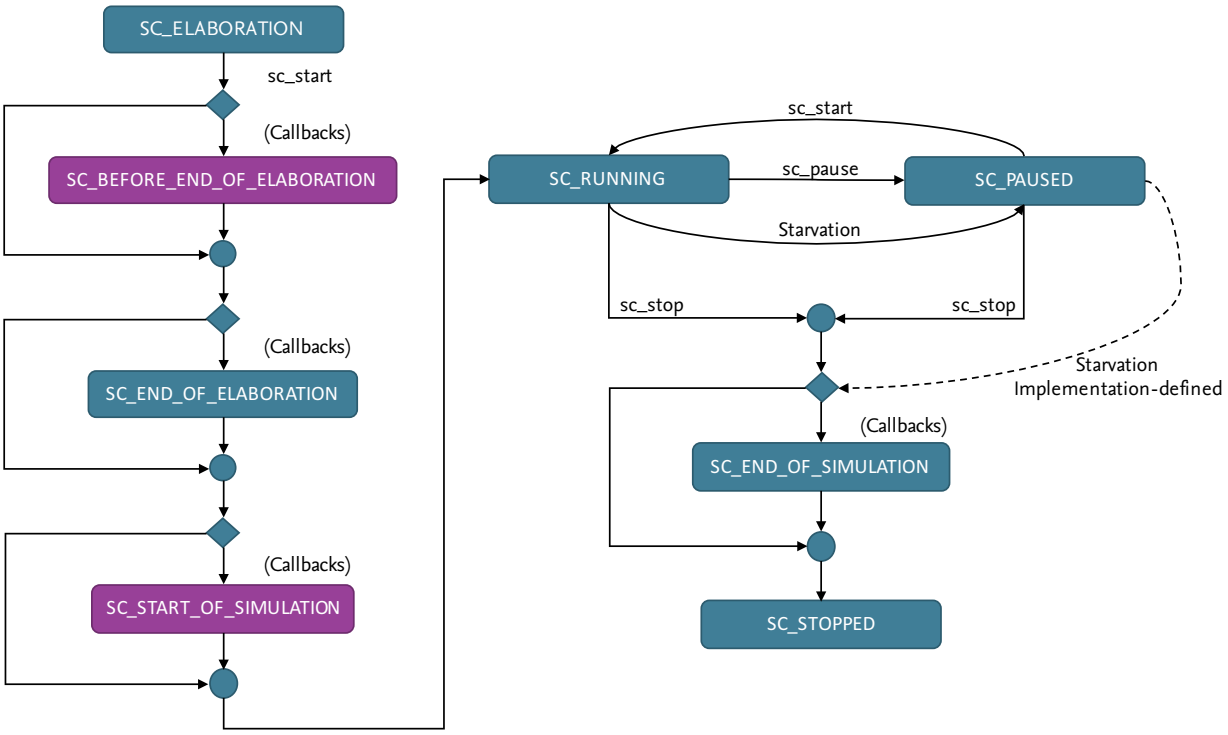

Figure 5.5: Extended SystemC phases adapted from [124]

SystemC version 2.3.1 introduced extended simulation phase callbacks as experimental feature. Figure 5.5 shows the state diagram for all simulation states. Decision points show were callbacks are possible between state changes. This SystemC kernel feature for multiple phase callbacks was chosen as facility for the snapshot manager to hook itself into the different simulation phases and fulfil its tasks. The two highlighted (violet) states are the state on which the snapshot manager will operate and load modules and module states.

The snapshot manager class needs to be able to run a callback function at certain simulation phases; the `SC_BEFORE_END_OF_ELABORATION` phase and the `SC_START_OF_SIMULATION` phase. During the elaboration all the simulation model objects are created. That is a good point to restore models that have integrated snapshotting functionality and were saved using the decorator pattern. The start of simulation phase is used to restore the internal state of already loaded models. Here we can also reset the simulation time for the restored simulation.

Again, the actual implementation of the phase callbacks is presented in the next Chapter in Section 6.2.2.

5.6 Accessing the SystemC hierarchy

In the previous Section, I already explained how checkpointable models can be marked for checkpointing with a decorator template. We also want to support simple checkpointing for other models, that don't come with their own checkpointing functions.

The platform has a list of all objects (modules) that need to be archived, so we can use SystemC kernel functions to discover models and their sub objects that need to be stored.

The USI delegation method uses the same mechanism for hierarchy traversal as will be used inside the snapshot manager. So I will describe it shortly.

Interface delegation provides a unified way to access simulation objects through the *scripting interface* via their hierarchical paths. This process is illustrated in Figure 5.6 with the construction of a delegation object for an `sr_register` instance. The process is broken down into six steps:

1. All simulation objects are created during the initialization phase. This of course also includes our example `sr_register` "obj.reg.ctrl"

2. The Python function `usi.find` is used to find our example object in the simulation context. This function also supports wild cards for hierarchical names.

3. Searching the requested `sc_object` instance by internally matching against all names in the current simulation context.

4. When a matching instance has been found, a new `USIDelegate` instance is created. The `USIDelegate` class is partly created in C++ and the target scripting language, in this case Python. The class has almost no functionality of its own. Its main task is to convert between `sc_object` and its registered interface class.

5. The *delegation kernel* tries converting `sc_object` to each registered interface class, in this example `sc_object`, `sr_register` and `AHBDevice`.

6. An `sc_object` will be dynamically cast to the desired interface. The cast creates a new SWIG proxy object if the interface is implemented for the specific `sc_object`.

After successful construction an `USIDelegate` object can be uses like any other normal Python object. All function calls are delegated transparently through the SWIG proxies.

Registering *Plug-in APIs* is only possible when they provide an object interface on an `sc_object` subclass, see also [117]. The registration process uses C++ macros and follows the *interface pattern*: *"A class can be inherited from any number of different interfaces to implement the corresponding functions. This allows others to cast the object to an implemented interface and use the provided functionality, regardless of how or what the object is or does."* [29, 117]

More technical details and example code can be found in [29] and [27]. The USI Python reference implementation is available on Github [125].

From the above described interface delegation process, the hierarchy traversal is most interesting for the snapshot manager class.

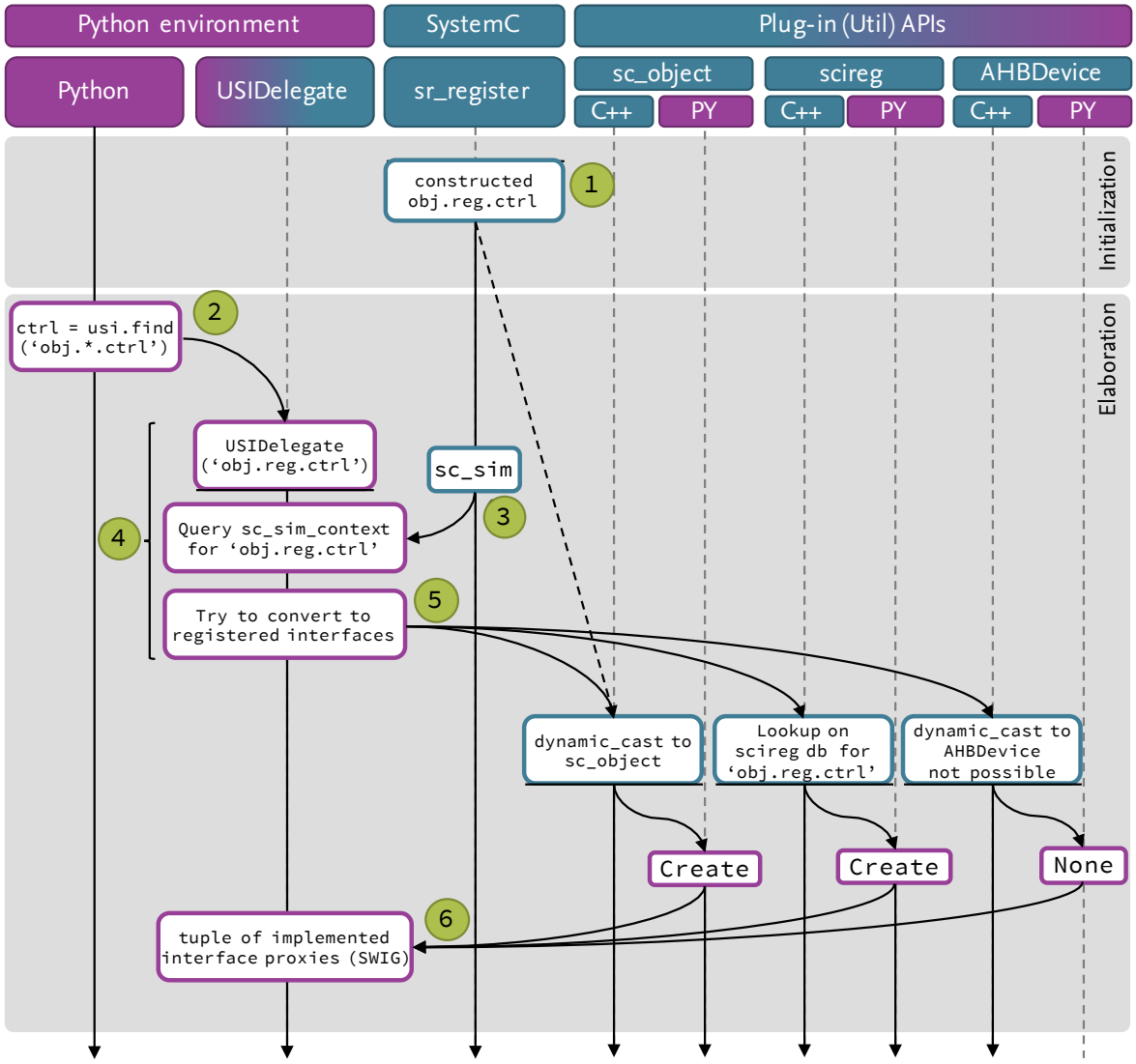

Figure 5.6: USI interface delegation example from [29]

Since we already store some models with their explicit checkpointing functionality, we can skip scanning for those and focus on models using the `sr_register` and `sr_signal` classes. Theses classes have been extended with checkpointing functionality, so any model using them has automatic checkpointing for its registers and signals.

This should be enough for most models to restore their state. More complex models, that store internal state apart from registers and signals, can implement custom checkpointing functions and be marked with the decorator template during instantiation. When the model code is not modifiable or when there is no desire to modify the code, the extensibility of the Cereal library can be utilized to create a custom extension to handle the model class. This way also legacy models can be made serializable without much effort.

Utilizing the extensibility of the Cereal library fits also with the requirements of transparency (R1) and self-reliance (R8). The model-library maintainer can take care of providing the necessary serialization extension while users and model developers can use their models as before.

In the next Section, I will look into necessary modifications on framework components for supporting serialization.

5.7 Integration of Serialization in SystemC

Serialization requires data members to be accessible from other objects or the class providing functions to interact with its private and protected data members. The `sr_signal` implementation used in this thesis already had a function in place for reading the current value of every type of signal, but there was no way to set the value directly of any kind of signal without causing side-effects in the connected modules. This functionality is required to safely implement save and restore functionality for signals.

The register implementation shipped with SoCRocket already contains side-effect-free reading and writing of stored register values, so there is no need for modification.

CCI parameters could also be used to save model state. The `sr_params` implementation in SoCRocket gives easy access to parameter values. Montón followed this approach in his thesis [84]. He relied purely on using the *GreenSocs* parameter implementation and had to specify every member variable he wanted to be snapshotable as `gs_param`. This approach is quite inflexible and severely inhibits the model developer. As long as these parameters are not a part of the standard this is not an option. This would also pollute the parameter space. Normally parameters should describe configurable parts of the model and not internal state. This would make changing actual configuration parameters non-straightforward and too complicated.

The necessary changes to the `sr_signal` implementation are detailed out in Section 6.1.2.

5.8 A note about virtual functions and templates

The snapshot manager class needs to be able to discover serializable models. This could be done with the hierarchy traversal and cast trick as is done in USI, but we would need to cast to the base class and not get the right type (model class) for the archive function of the Cereal library. We have to call the archive function of the derived class after all, since it knows about the class members.

An easy way to introduce serialization on a framework level would be to define pure virtual serialization functions inside a base class and force the developers to implement them in their models. The feasibility of that solution depends on how the serialization functions are implemented. The functions need to support all kinds of types and therefore are implemented with templates. Unfortunately C++ does not allow templated functions to be virtual. So the easy solution is out of the way. The Boost serialization library already had the same problem [126]. They advise to call the serialization function of the base class inside the serialization function of the derived class. Furthermore, it is recommended to make the serialization functions private and include an “access” class as friend, so that the serialization library can access the private function.

Most of the Cereal library interface is very similar to the boost serialization library. In fact Cereal can be seen as a modern C++11 header-only implementation of the boost serialization library. For this reason it has the same problem with inherited serialization functions.

The curiously recurring template pattern (CRTP) [127] could be one solution to achieve polymorphism without the reliance on virtual functions. Maybe using the curiously recurring template pattern for the base class and derivatives is an option. This technique uses a

templated base class. The derived classes use their own type as template argument for the base class. Using this technique would mean putting severe restrictions on a developer by not allowing the use of virtual functions and imposing another way of doing inheritance. This technique would be quite error prone.

Another approach could be to use the well-known decorator pattern [117] to extend existing model classes with serialization functionality. In the static version of the decorator pattern the newly created class is a templated class which is derived from its template argument. The base-class functionality can then be used as with any other base class and new functionality can be added on top. This option offers the most flexibility and is easily implemented outside of the existing model code.

Supporting full pointer and reference serialization would be quite complex and would severely impact performance of the serialization process. The boost serialization library does support it, but it is slow and not yet adapted to the C++11 standard. Properly saving a pointer involves checking the correct type of the object the pointer points to. The pointer can be a base class pointer, but the object can be of a derived class type. Several pointers could point to the same object. Each object should only be saved once. Same goes for restoring. The library has to be able to detect if it already created an object that the pointer needs to point to.

We can use the decorator pattern to have the decorator class create a list of serializable model classes in the form of smart pointers inside the snapshot manager instance. The list of serializable models does not need to be static, since there will be only one snapshot manager instance. The save and restore functions of the snapshot manager can then iterate over the vector of models. The decorator class definition is located in the `snapshot_manager` header and is displayed already in Figure 5.2 as an associated class with the *decorator* stereotype.

5.9 Summary

In this Section, I have presented my concept for how to extend a SystemC simulation framework with snapshotting functionality in a way that is portable and transparent to the user.

A serialization library that fitted my requirements formulated in Section 4.6 has been selected. This library decision influenced several design decisions. The snapshot manager class is designed in a way that fits with the SystemC standards while at the same time including the Cereal serialization library which relies on the C++11 standard. The portability of the library will make the snapshot manager implementation fairly straightforward, as will be seen in the next Chapter.

The Cereal library still lacks support for several crucial SystemC data types, which will be alleviated through library extensions that overload library functions that handle the saving and loading processes. This concept was presented with the example of `sc_time`. Working with the internal SystemC simulator time also requires access to private and protected data members of SystemC kernel classes. A concept for this type of access has been presented in Section 5.3.

Necessary SystemC kernel and framework features were identified that will help with the snapshot manager class implementation. At least one framework extension, `sr_signal`, will need to be slightly modified to support side-effect-free serialization.

Finally, the concept for SystemC model discovery and transparent inclusion of snapshot-

ting functionality in existing models was presented. All these concept will be implemented in the next Chapter.

6 Implementation

In the previous Chapter, I explained my architectural concepts and design decisions for the implementation of SystemC snapshotting. Moreover, I found a serialization library that fits with my requirements and detailed out C++11 concepts that will help me in implementing the snapshotting concepts in the snapshot manager class.

Before going into detail of the snapshot manager class implementation, some prerequisites from the used simulation framework have to be explained as they will be a crucial part in the snapshotting framework implementation and its examples.

First, I will provide a closer look at the SoCRocket register and TLM signal implementations. With those explanations being out of the way, I will focus multiple Sections on the actual implementation of the snapshot manager class and how it is integrated in the SoCRocket framework. Having described the snapshot manager implementation in detail, I will show a very minimal working example that I use to exercise the snapshot manager and verify that the snapshotting process works as I designed it.

6.1 SoCRocket

The whole snapshot framework would be much more basic without the SoCRocket SystemC simulation framework. SoCRocket already come with great standards compliant register implementation, signal implementation and a custom scripting interface. All of which will be described in the following Sections. These Sections correspond to the Sections in the architecture description.

6.1.1 SoCRocket Register Implementation

Register implementations for virtual platforms are often vendor-specific. Currently, there is no standardized way of modelling registers with SystemC/TLM. Cadence proposed interfaces for register introspection [128]. This proposal has not made its way into the standard yet. Another approach was made by GreenSocs with their GreenLib library that also contains GreenReg for easy register modelling [129]. Previous versions of SoCRocket relied on GreenReg for its register models.

The current SoCRocket register implementation `sr_register` has a very similar interface to GreenReg, so it can be used as a drop-in replacement. Furthermore, it supports the proposed `scireg` interface by Cadence. Even while maintaining compatibility with these interfaces `sr_register` has a much smaller code footprint than the original GreenReg.

The main advantage of `sr_register` is its usability and the lack of external dependencies. Listing 6.1 shows a minimal example of how to instantiate a register bank in a SystemC model.

Apart from register definition using address and data types, the function `init_registers` and optionally callback functions need to be declared. The register bank exposes functions to create registers, register fields and callbacks. An example register creation with two fields and a post-write callback can be seen in Listing 6.2.

The `sr_register` functions `create_register`, `create_field` and `callback` all return a

```
#include "sr_register.h"

class Device : public sc_core::sc_module {
    public:
        sr_register_bank<ADDR_TYPE,DATA_TYPE> reg_bank;
        void init_registers();
        void post_write_callback();
}
```

Listing 6.1: Instantiating an `sr_register` register bank

```
Device::Device() :
    reg_bank("register bank name") {
    Device::init_registers();
}

Device::init_registers() {
    reg_bank.create_register("register_name", "A Human readable
       description of the register",
          ADDRESS_OFFSET, DEFAULT_VALUE, BUS_WRITE_MASK)
        .callback(SR_POST_WRITE, this, &Device::post_write_callback)
        .create_field("six_bit_field", 6, 0)
        .create_field("single_bit", 18, 18);
}
```

Listing 6.2: Creating a register within the register bank

reference to their corresponding register. This makes it possible to chain the function calls. The register bank exposes the functions `bus_read` and `bus_write` to support connecting it to a bus. The bus interface is then responsible for handling error cases and delay. After creation registers and register fields can be accessed like a normal array. The source code for `sr_register` is available on Github [130].

6.1.2 SoCRocket TLM Signal

In Section 5.7, I have established the need to modify the SoCRocket TLM signal implementation. First, we will have a look how the signals work and then I can show which parts need to be extended to support side-effect-free reading of the data stored in the signals.

Standard TLM does not directly support the modelling of signals like interrupt lines. This would defy the purpose of abstracting away low-level communication. Nevertheless, it is sometimes necessary to have TLM-style signal communication. Usually, signal communication is then modelled using SystemC signals (`sc_signals`). SystemC signals are designed for accurate modelling of RTL signals. The SystemC kernel needs to schedule every read and write, though, and this would slow down a TLM simulation tremendously. Hence, SoCRocket comes with its own signal implementation `sr_signal`. A proposal from Cadence exists for a signal-wire implementation, but it has much less features than the

SoCRocket implementation [128].

The SoCRocket TLM signals are based on function-call communication, but retain the modelling style of the standard SystemC signals. Signal transmission is performed by directed function calls, similar to TLM blocking transports, but without the need to handle payloads.

A model that shall use the `sr_signal` signals simply needs to include the header file and call the `SR_HAS_SIGNALS` macro. The macro registers the model and creates certain utility functions to enable connecting signals between models and handling signal types correctly. This happens completely transparent to the user. In Listing 6.3, a module with an outgoing signal port of type *int* is shown.

```
1  #include "sr_signal.h"
2
3  class source : public sc_module {
4
5    public:
6      SR_HAS_SIGNALS(source);
7      SC_HAS_PROCESS(source);
8
9      signal<int>::out out;
10
11     // Constructor
12     source(sc_module_name nm) : sc_module(nm), out("out") {
13       SC_THREAD(run);
14     }
15
16     void run() {
17       // ...
18       out = i;
19       // ...
20     }
21 }
```

Listing 6.3: SystemC module with output signal from [131]

The actual signal is defined in line 9. In line 18, the variable `i` is written to the output. Alternatively to direct data assignment, the signals also provide a `write()` function. The `sr_signal` version used in this thesis was modified to provide a `read()` function as well.

Listing 6.4 shows a module with signal receiver.

The signal handler function `onsignal` is registered to the input signal “in” in line 11. If any function call is received on this signal, the function will be triggered. The data transmitted on the signal line can be accessed with the `value` variable. If the signal is not an integer type, the `onsignal` function has to be adjusted accordingly.

The connection of sender and receiver with the help of the `connect` function is shown in Listing 6.5.

```
#include "sr_signal.h"

class dest : public sc_module {

  public:
    SR_HAS_SIGNALS(dest);

    signal<int>::in in;

    // Constructor
    dest(sc_module_name nm) : sc_module(nm), in(&dest::onsignal, "in") {
    }

    // Signal handler for input in
    void onsignal(const int &value, const sc_time &time) {
      // do something
    }

}
```

Listing 6.4: SystemC module with input signal from [131]

```
#include "source.h"
#include "dest.h"

int sc_main(int argc, char *argv[]) {
    source src;
    dest dst;

    connect(src.out, dst.in);

    // ...

    return 0;

}
```

Listing 6.5: Connecting sender and receiver modules from [131]

The `connect` function is not only capable of this trivial direct connection, it can also perform broadcasting and multiplexing. Moreover, it can be used to convert between `sr_signal` and `sc_signal`. For broadcasting an output signal can directly be connected to multiple input ports. Multiplexing is possible when multiple transmitters are combined into one receiver. It is also possible to assign a channel number to a transmitter for easier identification.

In some cases, it might be necessary to connect an input signal to multiple output signals. For this use case, `sr_signal` provides the `infield` signal type. In contrast to a regular input signal, the infield signal carries knowledge about the source of the arriving signal. Each output signal can be bound to an infield channel. Each channel has its own value and the callback functions are able to identify the sender.

More technical details can be gained from the source code which again is available on Github [131].

Saving and Restoring Signals

Adding the side-effect-free read-functionality requires looking into the implementation in more detail. The macro `SR_HAS_SIGNALS` creates signal typedefs with the calling module class as template parameter. The code snippet can be seen in Listing 6.6. This means that every `sr_signal` type is directly linked to the module name where it was created. This complicates creating a unified `set_value()` function for all signals. Signals of type "out" already have a `write()` function but it is not free of side-effects.

Looking at the class structure in Figure 6.1 we see that every basic signal type class has a corresponding signal interface class. These specific signal interface classes all are derived from one virtual signal interface class `signal_if`. This class contains the basic read functionality needed by every signal so it makes sense to add the basic write functionality to this class as well. The function is listed in Listing 6.7. It has been included already in the class diagram in bold font face. The signal value `m_value` is defined as protected in the class and `TYPE` is a template parameter replaced by the requested signal type as shown also in 6.6.

```
#define SR_HAS_SIGNALS(name) \
        template<class TYPE> \
        struct signal { \
                typedef sr_signal::signal_in<TYPE, name> in; \
                typedef sr_signal::signal_out<TYPE, name> out; \
                typedef sr_signal::signal_inout<TYPE, name> inout; \
                typedef sr_signal::signal_selector<TYPE, name> selector; \
                typedef sr_signal::signal_infield<TYPE, name> infield; \
        };
```

Listing 6.6: `SR_HAS_SIGNALS` macro code

```
/// Setting value without side-effects
void set_value(TYPE value)
{
    m_value=value;
}
```

Listing 6.7: `set_value` function definition

Adjustments like this are not only beneficial to serialization of SystemC models, but also useful for injecting faults into running simulations. The injection of faults should only cause side-effects by the injected fault and not already by the function injecting the fault.

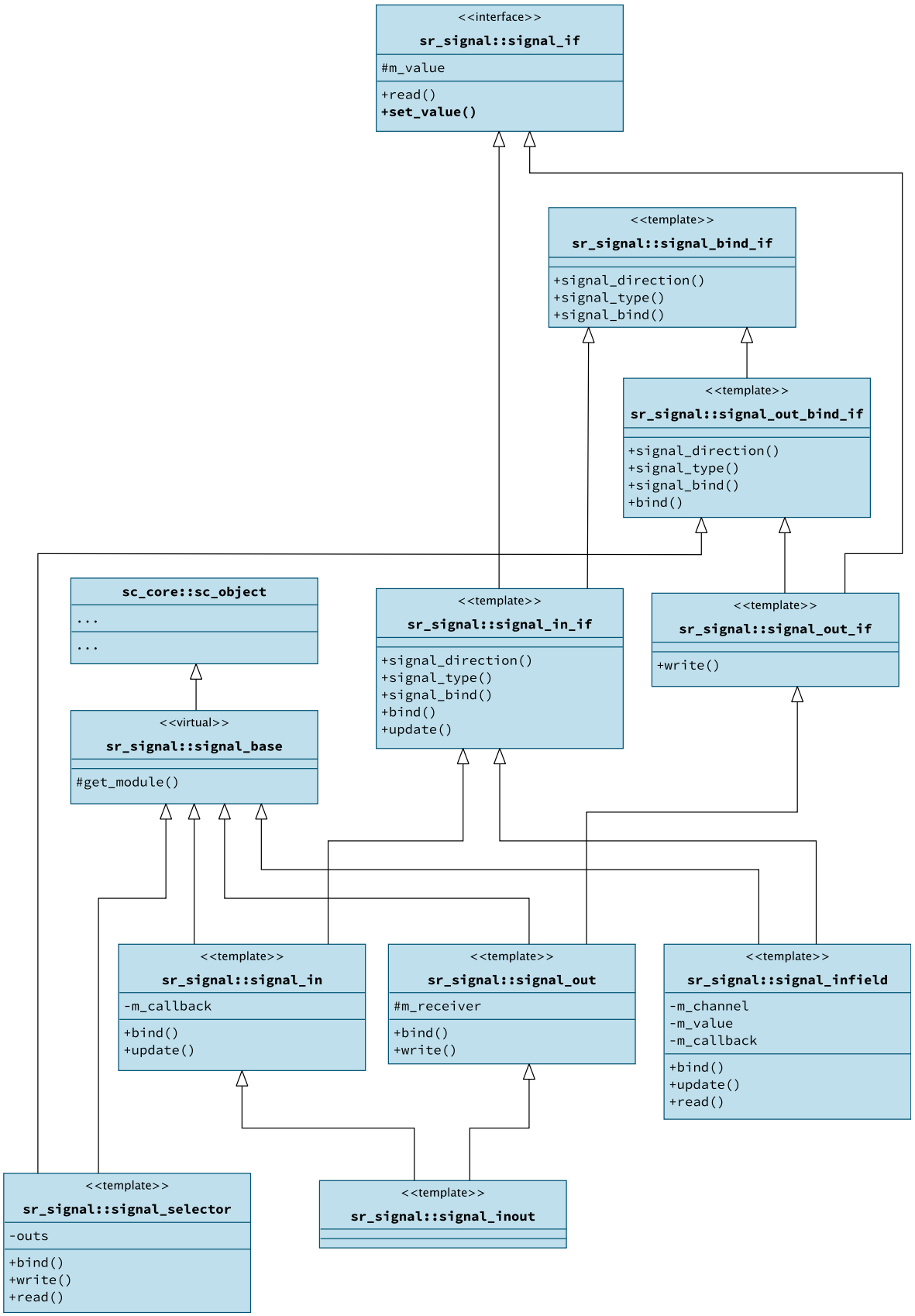

Figure 6.1: Class diagram for the `sr_signal` extension

6.2 Snapshot Manager Class

In Chapter 5, I have covered all conceptual topics regarding managing the snapshotting process and what should constitute the snapshot manager class. In this Section, I will focus on the various implementation details that comprise the snapshot manager class. First, I will revisit the topic of private member access and show how the concept of non-intrusive serialization presented earlier in Section 5.3 looks when it is implemented as part of a SystemC simulation framework. Then I move on to describe how the snapshot manager hooks itself into the SystemC simulation phases. Hooking into the simulation phases leads to accessing members of the SystemC hierarchy in the subsequent Section. The overall Section is concluded by a description of the serialization and storage implementation using the Cereal serialization library as designed in Section 5.2.

6.2.1 Accessing private members

In Section 5.3, I described the architectural aspects of accessing private and protected class members. In this Section, we will discover how the requirement of transparency was met in the implementation through private member accessing methods.

With the knowledge about C++11 advances from the previous Chapter, we can now look at the macros defined in Listing 6.8.

The two macros `CONCATE_` and `CONCATE` are helper macros to concatenate strings inside other macros. This functionality is usually handled through the wrapping of two macros as is shown here in the first two lines.

The `ALLOW_ACCESS` macro creates all the necessary structures to be able to access private members of a class. It takes the desired class, the private member type and private member name as arguments. The first templated struct defined in lines 5 and 6 defines a friend function which can be called using the ADL mechanism described above. Lines 7 and 8 define a utility class that define our desired private member type as friend and thereby allow the ADL mechanism to discover it. The class defined in line 9 is a specialization of the template classes from the previous lines which creates the necessary relationship between the two. Finally, lines 11 and 12 define the `ACCESS` macro which can be used to modify private members of class objects which have been allowed access. The arguments are a class object and the desired member variable.

In Listing 6.9 we can see an expanded version of the `ALLOW_ACCESS` macro shown in Listing 6.8. The shown code is used to access the `m_curr_time` member of the `sc_simcontext` class to save and restore the current simulation time.

The code in Listing 6.10 shows the expanded version of the `ACCESS` macro defined in Listing 6.8. The code is used to restore the simulation time to the value stored inside the snapshot, which is corresponding to the variable `loadtime` in this case.

This solution fits the requirement of enabling transparent snapshotting without modifying the SystemC kernel. With access to the header files it also allows creating snapshotting extensions for any binary-only models. A model developer now can choose which option to use. For legacy and proprietary code it is easier to write a Cereal extension and use the above described method to access any private members. For newly developed models it might make more sense to include checkpointing functions directly in the models. Later on, I will explain how the snapshot manger is able to handle both variants.

```
#define CONCATE_(X, Y) X##Y
#define CONCATE(X, Y) CONCATE_(X, Y)

#define ALLOW_ACCESS(CLASS, TYPE, MEMBER) \
  template<typename Only, TYPE CLASS::*Member> \
  struct CONCATE(MEMBER, __LINE__) { friend TYPE (CLASS::*Access(Only*)) { return Member; } }; \
  template<typename> struct Only_##MEMBER; \
  template<> struct Only_##MEMBER<CLASS> { friend TYPE (CLASS::*Access(Only_##MEMBER<CLASS>*)); }; \
  template struct CONCATE(MEMBER, __LINE__)<Only_##MEMBER<CLASS>, &CLASS::MEMBER>

#define ACCESS(OBJECT, MEMBER) \
  (OBJECT).*Access((Only_##MEMBER<std::remove_reference<decltype(OBJECT)>::type>*)nullptr)
```

Listing 6.8: Macros for private member access

```
template<typename Only, sc_core::sc_time sc_core::sc_simcontext::*Member>
struct m_curr_time57
{
  friend sc_core::sc_time (sc_core::sc_simcontext::*Access(Only*))
  {
    return Member;
  }
};

template<typename>
struct Only_m_curr_time;

template<>
struct Only_m_curr_time<sc_core::sc_simcontext>
{
  friend sc_core::sc_time (sc_core::sc_simcontext::*Access(Only_m_curr_time<sc_core::sc_simcontext>*));
};

template struct m_curr_time57<Only_m_curr_time<sc_core::sc_simcontext>, &sc_core::sc_simcontext::m_curr_time>;
```

Listing 6.9: Expanded `ALLOW_ACCESS` macro for time

```
(*sim).*Access((Only_m_curr_time<std::remove_reference<decltype(*sim)>::
   type>*)nullptr) = loadtime;
```

Listing 6.10: Expanded ACCESS macro for time

6.2.2 Enabling Extended Phase callbacks

Just like enabling C++11 support was needed for the Cereal library, the build procedure for the SystemC library needed to be adapted to enable this experimental feature. This was achieved through adding the environment variable `SC_ENABLE_SIMULATION_PHASE_CALLBACKS` in `core/waf/systemc.py` to the compilation environment and adding `-enable-phase-callbacks=yes` to the configure step.

Once the extended phase callbacks feature is activated any `sc_object` can register callbacks functions for the phases listed in Listing 6.11.

```
enum sc_status
{   // sc_get_status values:
    SC_UNITIALIZED=0x00,                // initialize() not called yet

    SC_ELABORATION              = 0x01, // during module hierarchy
       construction
    SC_BEFORE_END_OF_ELABORATION = 0x02, // during
       before_end_of_elaboration()
    SC_END_OF_ELABORATION       = 0x04, // during end_of_elaboration()
    SC_START_OF_SIMULATION      = 0x08, // during start_of_simulation()

    SC_RUNNING                  = 0x10, // initialization, evaluation or
       update
    SC_PAUSED                   = 0x20, // when scheduler stopped by
       sc_pause()
    SC_STOPPED                  = 0x40, // when scheduler stopped by
       sc_stop()
    SC_END_OF_SIMULATION        = 0x80, // during end_of_simulation()

    // detailed simulation phases (for dynamic callbacks)
    SC_END_OF_INITIALIZATION    = 0x100, // after initialization
//    SC_END_OF_EVALUATION        = 0x200, // between eval and update
    SC_END_OF_UPDATE            = 0x400, // after update/notify phase
    SC_BEFORE_TIMESTEP          = 0x800, // before next time step

    SC_STATUS_LAST              = SC_BEFORE_TIMESTEP,
    SC_STATUS_ANY               = 0xdff
};
```

Listing 6.11: `sc_status` enum in SystemC kernel source

The registration works by simply calling the `register_simulation_phase_callback` function and specifying for which phases the callback should be registered. The registration for the snapshot manager class is shown in Listing 6.12.

```
this->register_simulation_phase_callback( SC_START_OF_SIMULATION |
   SC_BEFORE_END_OF_ELABORATION | SC_END_OF_INITIALIZATION );
```

Listing 6.12: Registration of phase callbacks in snapshot manager constructor

The class registering for phase callbacks furthermore needs to implement the `simulation_phase_callback` function. The one included in the snapshot manager class is shown in Listing 6.13. The `sc_get_status` function is used to evaluate the current simulation phase upon which a switch function can decide what should happen during the callback handling. In our case here, we either load modules that have integrated snapshotting functionality and were stored using the decorator pattern above or we overwrite the state of existing modules that don't have integrated snapshotting functionality during the start of simulation phase.

```
void simulation_phase_callback()
{
    std::cout << sc_core::sc_get_status() << std::endl;
    switch(sc_core::sc_get_status())
    {
        case SC_BEFORE_END_OF_ELABORATION:
            load_modules();
            break;
        case SC_START_OF_SIMULATION:
            load_state_for_modules();
            break;
    }
}
```

Listing 6.13: Phase callback function in snapshot manager

6.2.3 Accessing the SystemC Hierarchy

The method of hierarchy traversal was already explained in detail in the previous Chapter. In this Section, I will show how this particular part is implemented within the snapshot manager class.

The registers and signals are stored in respective vectors inside the snapshot manager class. The save and load functions both use the recursive `scan_hierarchy` function shown in Listing 6.14. Since SystemC version 2.3 accessing the `sc_sim_context` object directly was deprecated, the `sc_get_top_level_objects` and `get_child_objects` had to be used to get a picture of the module hierarchy.

Using the hierarchy traversal ensures that the discovered registers and signals are saved in the correct order and can be restored to the same hierarchy layout during restoration.

During restoration the same `scan_hierarchy` function is used and the discovered register and signal objects are overwritten with the data from the snapshot file.

```
1  void scan_hierarchy(sc_object *obj)
2  {
3      const std::vector<sc_object*> *children = &obj->get_child_objects();
4      for ( unsigned i = 0; i < children->size(); i++ ) {
5          sc_core::sc_object *childnode = children->at(i);
6          if ( childnode ) {
7              sr_register<uint8_t> *reg = dynamic_cast<sr_register<uint8_t>
                   *>(childnode);
8              sr_signal::signal_if<bool> *signal = dynamic_cast< sr_signal::
                   signal_if<bool> *>(childnode);
9              if( reg ) {
10                 registers_vector_uint8.push_back(reg);
11             }
12             if (signal)
13             {
14                 signals_vector_bool.push_back(signal);
15             }
16             scan_hierarchy( childnode );
17         }
18     }
19 }
```

Listing 6.14: `scan_hierarchy` function of snapshot manager class

6.2.4 Serialization and Storage

The basic serialization functionality of the Cereal library was evaluated with the examples provided in the library's documentation. When basic functionality and a working example was established, the library was integrated into the SoCRocket SystemC framework.

Integration into the SoCRocket starts with adjusting the *waf* build system. Since the Cereal library is provided as headers and does not need compilation it can simply be checked out from its Github repository. Enabling it inside a simulation requires the whole build process to be adjusted to support C++11.

The standard compiler flags in the custom *waf* script `core/waf/flags.py` were adjusted to include `-std=gnu++11`. The GNU C++11 standard was chosen to maintain compatibility with already integrated libraries and models. Compiling the standard SoCRocket example platform with C++11 support worked just fine, so this adjustment can be regarded as safe and further development can be be done with C++11 features in mind.

The full *waf* build script for the Cereal library can be found in the appendix 10.3. The script checks for a local installation of the library and if it does not find any will checkout the repository from Github. Furthermore, it provides a so called `uselib_store` with which platforms can specify that they depend on the Cereal library for compilation.

```
1  #! /usr/bin/env python
2  # vim : set fileencoding=utf-8 expandtab noai ts=4 sw=4 filetype=python :
3  top = '../..'
4
5  def build(bld):
6
7      bld(
8          target       = 'counter.platform',
9          features     = 'cxx cprogram pyembed',
10         source       = 'sc_main.cpp so_main.cpp',
11         includes     = '.',
12         use          = ['BOOST', 'usi',
13                         'srcount', 'srsequencer',
14                         'sr_registry', 'sr_register', 'sr_report', '
                               sr_signal', 'common',
15                         'TLM', 'SYSTEMC', 'CEREAL'
16                        ],
17     )
```

Listing 6.15: Build script for minimal example platform

The minimal example platform described in Section 6.4 needs Cereal to work properly. The platforms build script is shown in Listing 6.15. In line 15 the usage of the Cereal library is specified.

With the library properly integrated into the framework, the actual integration of serialization features can commence.

JavaScript Object Notification

JSON is, contrary to its name, a language-independent data format and widely used as a data-storage and communication format for web applications. An example can be seen in Listing 6.16. The Listing shows how little text overhead JSON has, which makes it very readable and fast to parse.

In SoCRocket, the JSON format is used for parameter configuration files. Through these external configuration files it is possible to define the simulation configuration without the need for recompilation of the platform. JSON interpreters exist for a multitude of languages. A quite efficient interpreter implemented in C++ is RapidJSON [132], which is included in the serialization library I am using in this work.

6.3 Making serializable models discoverable

In Section 6.2.3, I described how the snapshot manager class can traverse the SystemC hierarchy and discover models. This works mostly for models where the type is known. There might also be models that are not yet known, but should be snapshotted as well. These could be models that are dynamically created through a factory for example. Therefore, in Section 5.8 I described the architectural concept behind the decorator design pattern. Using this design pattern a decorator class can be implemented that can be attached to existing

```
{
    "First Line": "One",
    "Index":
    {
        "value" : 1
    },
    "Second Line": "Two"
}
```

Listing 6.16: JSON Example

classes upon instantiation to wrap a certain structure around them.

Listing 6.17 shows the code for the *serializable* decorator class. The resulting class will be derived from the template parameter class. It simply contains a constructor that calls the constructor of the original class with some extra functionality. In our case this extra functionality consists of creating a unique pointer pointing to the instantiated model and registering that same pointer with the snapshot manager class.

```
template <class T>
class serializable : T
{
public:

    serializable(sc_core::sc_module_name mn)
        : T(mn)
    {
        std::unique_ptr<sc_core::sc_module> ptr(this);
        snapshot_manager::module_vector.push_back(std::move(ptr));
    }
};
```

Listing 6.17: Decorator class for serializable models

In this case, the `unique_ptr` was chosen to store a reference to the derived class object, because the Cereal serialization library offers support for polymorphism and smart pointers. The Cereal library uses run-time type information to determine the true type of a polymorphic base class pointer.

Since C++11 offers only rudimentary introspection and reflection support, the Cereal library needs to know about the relationship between derived and base class during compilation time. This relationship can be registered with helper macros either in the derived class header or its implementation. The base class does not need to have serialization functions, but needs to be known to the Cereal library. In case of the *serializable* decorator implemented here, the Cereal library needs to know about the relationship of the model made serializable to the `sc_module` base class.

As all SystemC models are derived from the `sc_module` base class, the class needs to be registered with the Cereal library. However, the `sc_module` base class is an empty base class.

The Cereal library checks for empty classes when using Extensible Markup Language (XML) or JSON archives. Therefore, in the case of an empty base class, an empty serialize function has to be added. For binary archives empty classes are not an issue. Since I am using JSON archives for the snapshots an empty serialize function should be added to the `sc_module` class.

```
#include "cereal/archives/json.hpp"
#include "cereal/types/base_class.hpp"
#include "cereal/access.hpp"

#include <core/common/systemc.h>
#include <tlm.h>
#include <stdint.h>

namespace cereal
{
    template <class Archive> inline
    void serialize( Archive & ar, sc_core::sc_module const & mod )
    {
    }

    template <> struct LoadAndConstruct<sc_core::sc_module>
    {
    template <class Archive>
    static void load_and_construct( Archive & ar, cereal::construct<sc_core
        ::sc_module> & construct )
    {
    }
    };
}
```

Listing 6.18: Cereal extension to support `sc_module` base class

However, modifying the SystemC kernel code for the `sc_module` class is not an option, the implementation was done slightly differently. Listing 6.18 shows the Cereal extension that is able to support the `sc_module` class as base class for snapshotting generic SystemC modules derived from `sc_module`. The `CEREAL_REGISTER_POLYMORPHIC_RELATION` macro calls the access construct which needs a default constructor which is not available for all base classes. Since the `sc_module` base class does not have a default constructor, the templated function `load_and_construct` [133] needed to be added to the class. Modifying the `sc_module` base class in the Kernel would break with the transparency requirement of our serialization library. Fortunately, Cereal offers the templated `LoadAndConstruct` class which can be specialized with the class type that needs to be extended with the `load_and_construct` function. The `load_and_construct` function is then implemented within the `LoadAndConstruct` class. Both the `serialize` and the `load_and_construct` functions can have empty bodies, since we will never save or restore objects of the `sc_module` base class directly.

When a model is declared serializable by using the decorator template class, the model needs to implement a `serialize` as well as a `load_and_construct` function to save and restore its member variables. The `load_and_construct` function is necessary in case the developer did not use the default constructor and wants some extra functionality while constructing the module object.

6.4 Minimal SystemC Platform Example

To properly test the functionality of the snapshotting framework a rudimentary example platform is needed. The platform should allow testing all features while still being relatively compact. To achieve this, I have decided to design a small state machine that uses a single register as state memory.

The absolute minimal platform configuration is visualised in Figure 6.2. It comprises a SrSequencer module which is connected to a SrCount module via two signal lines of type sr_signal.

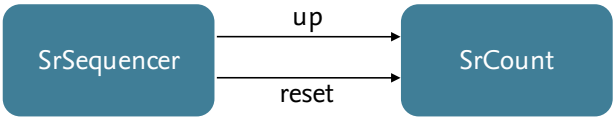

Figure 6.2: Minimal platform configuration

The SrSequencer module has two threads triggering the reset and up signals. The up signal is triggered every 20 nanoseconds. Triggering the reset signal takes place every 75 nanoseconds. Apart from this the SrSequencer has no further functionality.

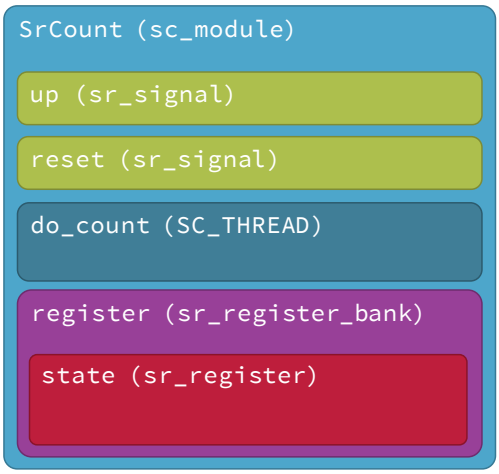

Figure 6.3: SrCount internal structure

The SrCount module is slightly more complex. Its internal components can be seen in Figure 6.3. The two signals, up and reset, correspond with the SrSequencer signals. Each signal has an associated function that is called, whenever the signal is triggered. The module state is represented with an internal variable and a register. The state register holds the current state of the internal state machine whose behaviour is depicted in Figure 6.4. The

internal variable “x” is initialized to the value 0 in the constructor and incremented by one in the endless loop inside the do_count thread. The do_count thread furthermore prints the current values of x and the state register every 5 nanoseconds.

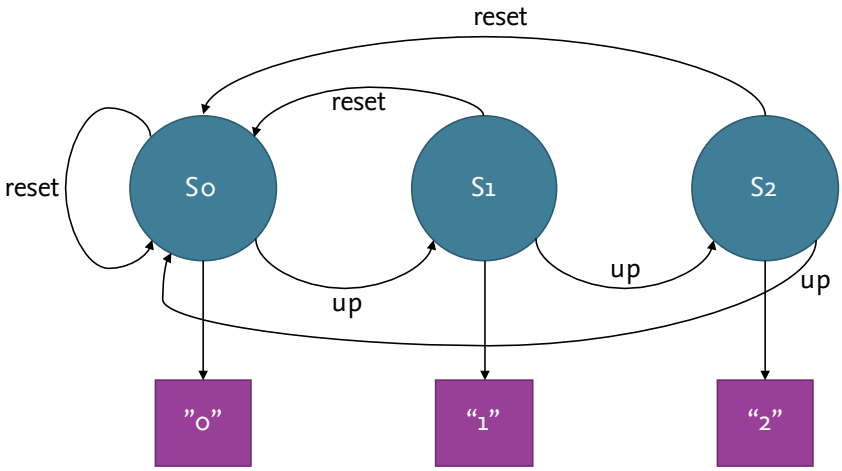

Figure 6.4: Internal state machine of module SrCount

This minimal platform allows evaluating the snapshotting process with various possible states in multiple locations. The state of internal variables that are not accessible via the SystemC hierarchy is represented by the counter variable x. Register state is covered as well as the internal state of the signal connecting SrSequencer and SrCount. Furthermore, the state machine logic can be adjusted before restoring from a snapshot. The proof-of-concept is limited to loosely timed modelling style, but the techniques evaluated here are also applicable for an approximately timed modelling style.

6.5 Summary

In have laid out the underlying concepts for this Chapter already in the previous Chapter. The previous Chapter focussed solely on the architectural aspects of the snapshotting process and the design decision I took. In this Chapter, I described how I implemented the previously presented concepts.

Before diving directly into the implementation of the snapshot manager class, I gave more insights into the inner structure of the SoCRocket register and signal extensions. The register implementation could be used as is. The SoCRocket TLM signal implementation was first analysed and subsequently extended for snapshotting. The internal signal values need to be readable and modifiable without any side-effects. This was achieved through modification of the basic signal interface class.

With the SoCRocket framework components prepared for snapshotting, the snapshot-manager-class implementation followed. The overall structure has already been described in Figure 5.2. In this Chapter, I focused on the four implementation goals for the snapshot manager class. First came the implementation of macros to allow for private class member access of other classes such as SystemC kernel internals. Second were the extended phase callbacks that were introduced recently in SystemC. They allow hooking the snapshot manager into multiple simulation phases to perform its tasks. Third was the implementation of

the hierarchy traversal function analogue to the one found in USI. Finally the implementation of the serialization functionality for storing of the snapshot data was implemented with the Cereal serialization library.

One thing was still missing from the snapshot manager class: the ability to discover models. This discovery mechanism was implemented using a decorator class that can be wrapped around SystemC modules during instantiation in the `sc_main` function.

The last Section covered the implementation of a very basic and minimal SystemC platform to exercise all the previously implemented functionality. The results of this exercising will be presented in the following Chapter.

7 Evaluation

In the previous Chapter, the snapshotting framework with UVM support was explained in detail. Some example code was shown, but not how everything is executed or how user friendly the framework is. In this Chapter, the framework performance will be evaluated and compared against *DMTCP*, which was presented already in 4.1.3.

Furthermore, I will show how the requirements formulated in Section 4.6 have been met. In Section 4.4, I have pointed out that testing and verification are important parts of Continuous Integration. In order to make use of my snapshotting solution in a CI workflow it has to play well with testing and verification frameworks. The most prevalent one is currently UVM.

The next Section will be devoted to describing the implementation and inclusion of the UVM SystemC library into the SoCRocket framework as well as relevant background knowledge about UVM.

The minimal platform example that is used for early snapshotting testing will be used in conjunction with the UVM framework to enable evaluation of my snapshotting framework against DMTCP. The metrics for this evaluation are described in Section 7.3. In the subsequent Sections the various evaluation measurement results are presented.

The minimal platform example is good to quickly enable evaluation and comparison of snapshotting functionality between different snapshotting solutions. It does not however represent a real world example of a SystemC simulation.

Since SoCRocket comes with a large model library, I picked Gaisler's interrupt request (IRQ) controller as a suitable device under test for the integrated UVM framework. Multiprocessor Interrupt Controller (IRQMP) is explained in Section 7.8. Snapshotting an IRQMP integrated in a SystemC platform together with my snapshotting framework and the UVM library shows how usable my framework is in real world use cases.

7.1 Extending the framework for UVM

In Section 3.5, I explained the basic concept of UVM and gave an overview of its essential components. The interested reader can find more details about the SystemC UVM library in Appendix 10.1. These components will now be integrated into the snapshotting framework. Until now the minimal platform has its own very simplistic sequencer for the counter module. In this Section, our SrCount module will be integrated as DUT in a proper UVM environment.

As can be deducted from the description in Section 3.5.1, the UVM class library needs various modes of communication between a large number of components. At least, simple forwarding and broadcasting mechanisms are needed. TLM offers these communication methods, which are already integrated into the latest SystemC standard. A basic concept underlying TLM is the separation of communication and function within modules. The communication details are abstracted so that many different protocols can be covered by TLM. This way, TLM is not dependent on a specific interface providing a specific protocol. Thanks to the standardized interfaces within the components, a simple interface exchange

is possible. This allows reusing the same test environment for different revisions of a DUT. The DUT can be first realised as a simple TLM model. As the model matures and gains more deeper abstraction levels, only a few changes need to be made to the classes defining the test environment. The test belonging to a DUT is consistent throughout the development cycle.

For the purpose of standardization, two TLM versions exist which are *TLM-1* and *TLM-2.0* [134]. Since the timing information of the communication is not relevant in the test environment and only simple forwarding of transactions is required, UVM uses mainly *TLM-1*. This leads to some complications which were addressed during the development of the UVM SystemC class library. UVM-SystemC has equivalent base classes and member functions to maintain UVM compatibility. Furthermore, existing SystemC functionality like the aforementioned TLM interfaces as well as reporting structure are used. Martin Barnasconi gave an overview of the similarities of TLM in standard UVM and UVM-SystemC in his 2016 DVCon Talk [100].

Regarding the communication interfaces between UVM components, I will not go into detail here. The interested reader is directed to the UVM user guide [42].

```
1  #include <systemc>
2  #include <uvm>
3  #include "core/common/sr_report.h"
4
5  #include "snapshotting/models/srcount/srcount.h"
6  #include "snapshotting/models/uvm_test/agent.h"
7  #include "snapshotting/models/uvm_test/testbench.h"
8
9  int sc_main(int, char*[])
10 {
11     sr_report_handler::handler = sr_report_handler::default_handler;
12     SrCount* counter = new SrCount("counter");
13
14     uvm::uvm_config_db<SrCount*>::set(0, "testbench.agent1.driver.*", "dut"
           , counter);
15     uvm::uvm_config_db<SrCount*>::set(0, "testbench.agent2.monitor.*", "dut
           ", counter);
16
17     testbench* tb = testbench::type_id::create("testbench");
18
19     uvm::run_test();
20
21     return 0;
22 }
```

Listing 7.1: Simplified `sc_main` for UVM simulation

As in any other SystemC simulation, the build-up of an UVM test environment starts with a `sc_main` function. Here, we will instantiate the DUT, in our case `SrCount`. In Listing 7.1, this happens in line 12. Our counter module does not need any special parameters, so the

module name is sufficient. Furthermore, the test bench itself is instantiated in line 17. The test bench uses the factory mechanism explained earlier, which is included with the UVM library. It is worth noting, that we do not explicitly connect the counter to the test bench. These connections are handled through the configuration database, which can be seen in lines 14 and 15. The two lines can be read as follows:

The pointer `counter` is made visible to the configuration database with the name "dut". It is available to all components within the component hierarchy below the components "agent1.driver" and "agent2.monitor" , which are themselves located in the component "testbench".

We will have a closer look, when I explain the *agent*.

```
1 class testbench : public uvm::uvm_env
2 {
3 public:
4     agent* agent1;
5     agent* agent2;
6     scoreboard* scoreboard0;
```

Listing 7.2: `testbench` with subcomponents

7.1.1 The *test bench*

The *test bench* inherits from the UVM library class `uvm_env`. As already explained earlier, it is a component that comprises the subcomponents needed for a test.

Listing 7.2 shows that the subcomponents are laid out as public members of the component hierarchy. The subcomponent instantiation however does not take place within the constructor. The instantiation of subcomponents makes use of the sequence of the different phases defined in UVM. Specifically, this happens in the build phase shown in Listing 7.3. As seen in Listing 7.2, the subcomponents are defined as pointers. This means in the build phase the factory mechanism can be used to construct the subcomponents within the hierarchy. In line 6 a new instance of an agent is created. A pointer to the test bench is passed as second parameter to the factory, which is one order higher in the hierarchy. Passing the pointer like this creates the component hierarchy, so that in the configuration database a speaking string such as "testbench.agent1" can be used. In line 11 the configuration database is used to configure "agent1" as "is_active" in order to use it as driving component for our DUT. Line 20 requires more attention. In these lines the configuration database is used to attach a specific sequencer to a sequence. The sequencer will start immediately after starting the run phase with the creation of test stimuli. The test stimuli are in turn created by the sequence. Since we only have one sequence in our test, this is the fastest way. Alternatively, a sequence can be started manually by calling its `start()` function. The next phase following the build phase is the connect phase in which the subcomponents and components are connected. In our *test bench* this means connecting *agent* and *scoreboard*. The macro `UVM_COMPONENT_UTILS(componentname)` is necessary in each component that shall be used with the factory to create the necessary infrastructure in it.

```
    void build_phase(uvm::uvm_phase& phase)
    {
        srInfo()("Build phase");
        uvm::uvm_env::build_phase(phase);

        agent1 = agent::type_id::create("agent1", this);
        assert(agent1);
        agent2 = agent::type_id::create("agent2", this);
        assert(agent2);

        uvm::uvm_config_db<int>::set(this, "agent1", "is_active",
            uvm::UVM_ACTIVE);
        uvm::uvm_config_db<int>::set(this, "agent2", "is_active",
            uvm::UVM_PASSIVE);

        scoreboard0 = scoreboard::type_id::create("scoreboard0", this);
        assert(scoreboard0);

        // alternative: starting manually by invoking seq->start()
        uvm::uvm_config_db<uvm::uvm_object_wrapper*>::set(this, "agent1.
           sequencer.run_phase", "default_sequence", sequence<dut_trans>::
           type_id::get());
    }
```

Listing 7.3: `testbench` UVM build phase

7.1.2 The *transaction*

The transaction derives from the class `uvm_sequence_item` which in turn derives from the class `uvm_transaction`. The latter provides a method to enable modification of the transaction id. This is a necessary feature for the communication via feedback between *driver* and *sequencer* towards the *sequencer* or the *sequence*. Furthermore, the `transaction` class contains data fields which can be translated into the communication protocol of the DUT. Listing 7.4 shows the fields of our transaction class.

```
public:
    uint32_t addr;
    uint32_t data;
    op_t op;
    sc_core::sc_time time;
```

Listing 7.4: `transaction` class data fields

The data field `op` stores the transaction type. The driver can then select the correct algorithm for translating the other data fields based on the transaction type. The other fields are explained when we get to the other components. Moreover, the transaction

class contains several helper methods which are not shown in Listing 7.4. One of those methods is `do_copy` which copies complete transactions or `convert2string` which helps with displaying transaction in a log. Another important function is `do_compare`, which compares two transactions. Depending on the operations type the function compares the corresponding data fields of the transactions. Thus proving essential in the verification process while comparing the actual value against the desired value of DUT output signals.

7.1.3 The *sequencer*

The sequencer does not need any additional functionality, therefore it is simply derived from the UVM class `uvm_sequencer<T>`. The template parameter `T` specifies the transaction class that the sequencer delivers. As the sequencer class is templated, the normal factory macro `UVM_COMPONENT_UTILS` is not sufficient here. If the sequencer is to be used with the factory the macro `UVM_COMPONENT_PARAM_UTILS(componentname<T>)` has to be used. In our example `T` is the class `dut_trans`.

```
1    void build_phase(uvm::uvm_phase& phase)
2    {
3        srInfo()("build_phase");
4        uvm::uvm_agent::build_phase(phase);
5
6        if(get_is_active() == uvm::UVM_ACTIVE)
7        {
8            srInfo()("Set to active mode.");
9            sqr = vip_sequencer<dut_trans>::type_id::create("sequencer",
                 this);
10           assert(sqr);
11           drv = vip_driver<dut_trans>::type_id::create("driver", this);
12           assert(drv);
13       }
14       else
15       {
16           srInfo()("Set to passive mode.");
17           mon = vip_monitor::type_id::create("monitor", this);
18           assert(mon);
19       }
20   }
```

Listing 7.5: `build_phase` of an agent

7.1.4 The *agent*

The agent comprises all components that are needed to communicate with the DUT. It contains a pointer to a sequencer, a driver and a monitor. All three subcomponents are instantiated with the factory during the build phase. As already described in Section 3.5.1, an agent can be used in two different modes of operation. In Listing 7.3 in lines 11 to 14, the agents operating mode is set using the configuration database during the `build_phase`

of the test bench, which is in this case the superordinate component within the hierarchy. Listing 7.5 shows the `build_phase` code of an agent.

In line 6 of Listing 7.5, we can see the usage of the `get_is_active()` method to find out which operating mode was set in the configuration database. The method is defined in the UVM class `uvm_agent` from which our agent class is derived. The function is merely a pretty wrapper to access the configuration database, which could have also been done manually.

7.1.5 The *driver*

As the name suggests, the driver is used to drive the DUT. It takes stimuli from the sequencer which come as transactions and translates them into the communication protocol of the DUT. The UVM user guide [42] suggests using a specially defined interface class for the communication between driver, monitor and DUT. This interface shall be deposited within the configuration database to which driver and monitor have access. In our case the DUT uses signals from the `sc_signal` library. This library can only be uses within SystemC modules. Therefore we cannot simply create an interface class that uses the `sc_signal` library. Another solution is depositing the DUT itself in the configuration database. Any interface class relying on DUT specific libraries needs to be adapted and is not universally usable for multiple DUTs. Furthermore, the driver and monitor depend on the DUT. Listing 7.1 shows in line 14 how our DUT `SrCount` is stored in the configuration database. The driver can access the DUT from the configuration database like shown in Listing 7.6

```
1        if(!uvm::uvm_config_db<SrCount*>::get(this, "*", "dut", counter))
2        {
3            UVM_FATAL(this->name(), "SrCount not defined. Simulation
                 aborted!");
4        }
```

Listing 7.6: Accessing a DUT stored in the configuration database

Access to DUT functions is possible by using the pointer `counter`. Furthermore, the driver needs to have signals to simulate communication with the DUT. These signals are shown in Listing 7.7.

```
1    typename signal<bool>::out up_out;
2    typename signal<bool>::out reset_out;
```

Listing 7.7: Interfaces for communicating with the DUT

The function `drive_up` provides the core functionality of the driver and therefore is shown in its entirety in Listing 7.8.

Communication with the *sequencer*

The driver does its main work during the `run_phase`. As with the other phases, this one is called automatically by UVM-SystemC. In this phase, the driver requests transaction objects from the sequencer in an endless loop. The process works analogous to the standard TLM get-port. The parent class of the driver, `uvm_driver<REQ>`, provides a `uvm_seq_item_pull_`

```
1    void drive_up(const REQ& req)
2    {
3        up_out.write(true);
4        wait(10, SC_NS);
5        up_out.write(false);
6        UVM_INFO(this->name(), "transmitting " + req.convert2string(), uvm
             ::UVM_MEDIUM);
7        wait(50, SC_NS);
8    }
```

Listing 7.8: Driver function for DUT communication

`port<REQ, RSP>` [135]. When the driver calls the function `get_next_item(req)`, the next transaction is delivered by the sequencer and stored in the variable `req`. The content of the data field `op` within the transaction decides how this transaction is further processed. Usually, this is achieved through a switch-case-statement and calling the respective functions of the driver. Listing 7.9 shows the transaction processing loop with the calls to the above described functions.

```
1        while (true) {
2            this->seq_item_port->get_next_item(req); // or alternative this
                 ->seq_item_port->peek(req)
3
4            srInfo()
5                ("transaction",req.convert2string())
6                ("Processing");
7            switch(req.op)
8            {
9                case UP_WRITE:
10                   drive_up(req);
11                   transaction_port.write(req);
12                   break;
13               case REG_READ:
14                   read_reg(req, rsp);
15                   rsp.set_id_info(req);
16                   break;
17               case RESET_WRITE:
18                   drive_reset(req);
19                   transaction_port.write(req);
20                   break;
21               default:
22                   break;
23           }
24           this->seq_item_port->item_done();
25           if(req.op == REG_READ)
26           {
```

```
27                  // is blocking; sequence must issue a corresponding
                        get_response
28                  this->seq_item_port->put_response(rsp);
29              }
30          }
```

Listing 7.9: Communication with the sequencer and further processing

Feedback to the sequencer or the sequence

Usually, a monitor is used to interface with the output of the DUT. In our case we can also use another option, as already mentioned above. We can set the id of the original request inside the response transaction and thereby linking them both.

When working with communication protocols that use both signals and transactions it makes sense to exfiltrate the transactions the driver sends to the DUT via an *analysis port*. This way they can later be compared easily without much translation login in-between. Theoretically any number of components could be used for processing of transactions from the analysis port. In our example this is handled by the scoreboard.

7.1.6 The *monitor*

The monitor is the counterpart to the driver. It monitors the DUT outputs and translates them into transactions. The interfaces shown in Listing 7.10 have the following functions:

```
1     signal<uint8_t>::in* state_in;
```

Listing 7.10: Monitor interfaces

The analysis port has the same functionality as in the driver. The signals received from the DUT are translated into transactions and forwarded for further processing. As with the driver, the processing is handled by the scoreboard.

Furthermore, the monitor interfaces can be extended with other signals in case the DUT has output signals, that need to be connected.

7.1.7 The *sequence*

The sequence can be seen as the heart piece of a test. It defines the stimuli for the DUT. The parent class `uvm_sequence<REQ, RSP>` provides the function `get_response` which is used to receive a response from the driver. This function blocks execution until it receives the requested response from the driver. So it is important to take care that the communication protocol between driver and sequencer does not stall. Therefore the functions `put_response` and `get_response` have to occur pairwise within driver and sequencer.

Additionally to the phases specified in Section 10.1.1 the sequence class has further phases which stem from deriving from the class `uvm_sequence_base`. These additional phases are accessible to the user through callbacks as well. The callback function `body` is of particular importance here as it contains the sequence implementation which is generating the stimuli. Figure 7.1 shows the newly added phases in their execution order. The phases `pre_body` and `post_body` are only executed automatically before and after the `body` phase by UVM when the sequence is started with the additional parameter `call_pre_post=true`.

Figure 7.1: The various phases of a UVM sequence according to [42]

The stimuli

As was mentioned above, the definition of test stimuli happens within the callback function `body`. To further illustrate their creation we will look at one example.

The SrCount module does not require any configuration steps, so we can directly start by stimulating on of its input ports. Listing 7.11 shows an excerpt from the `body` function. First, we create a request (`req`) transaction. The `addr` data field is not relevant for this test, so we can leave it at zero. The `data` field is set to the expected value the monitor will see at the output of the DUT. The operation type (`op`) is set to `UP_WRITE`.

The function `start_item` forwards the transaction object to the driver via the sequencer for immediate execution. It is also possible to pass a specific sequencer to the function. In our case we use the default sequencer. The sequence was linked to the sequencer in the configuration database. So we do not have to set the default sequencer again here. Furthermore, we could define a priority for the transaction. Our test bench does not contain an arbiter, so it does not make much sense to work with priorities.

The function `finish_item` completes the transaction. Between the two functions other functions could be executed, but they should not spend any delta cycles.

```
1        req = new REQ();
2        rsp = new RSP();
3        req->addr = 0;
4        req->data = 1;
5        req->op = UP_WRITE;
6
7        srInfo(this->get_full_name().c_str())("Send up");
8
9        this->start_item(req);
10       this->finish_item(req);
```

Listing 7.11: Sending an up signal in the UVM sequence

Now that we have stimulated our DUT to actually do something, we want to see if its internal state has changed as well. The SrCount module stores its state in a single register which can be accessed by other components.

The code for reading a register value looks very similar to our previous example. This time we will also get a response from the DUT and check immediately if it was correct. The corresponding code is shown in Listing 7.12. Again, we need to create a transaction for the DUT. Here the address field is relevant, although the DUT only has one register it is again 0. The data field is set to the value we want to see in the response. After calling the `finish_item` function we have some further steps now.

The `get_response` function is necessary here to ensure the correct processing of the transactions. As already explained in Section 7.1.5, we need a corresponding response transaction when processing a transaction with operation type `REG_READ`. If we would omit the call to `get_response`, the driver could not process further transactions and would cause the whole test to stall.

```
1         req = new REQ();
2         rsp = new RSP();
3         req->addr = 0x0;
4         req->data = 0x1;
5         req->op = REG_READ;
6
7         srInfo(this->get_full_name().c_str())("Check reg");
8
9         this->start_item(req);
10        this->finish_item(req);
11        this->get_response(rsp);
12
13        if(rsp->data != req->data)
14        {
15            std::ostringstream str;
16            str << "Error, address: 0x" << std::hex << req->addr;
17            str << " expected data: 0x" << std::hex << req->data;
18            str << " actual data: 0x" << std::hex << rsp->data << std::endl
                  ;
19            UVM_ERROR(this->get_name().c_str(), str.str());
20        }
21        else
22        {
23            std::ostringstream str;
24            str << "Success, address: 0x" << std::hex << req->addr;
25            str << " expected data: 0x" << std::hex << req->data;
26            str << " actual data: 0x" << std::hex << rsp->data << std::endl
                  ;
27            UVM_INFO(this->get_name().c_str(), str.str(), uvm::UVM_MEDIUM);
28        }
```

Listing 7.12: Reading register in the UVM sequence

After receiving the response transaction, a simple if-condition checks whether the resulting data field contains the expected value. The macros `UVM_ERROR` and `UVM_INFO` are used to display messages for the user. After termination of the test, UVM uses these message to create a statistic of displayed error and info messages per component.

The runtime phases of UVM are executed as independent, parallel processes. Therefore synchronization at certain points is unavoidable. The *objection* mechanism is one option provided by UVM for such cases. Prior to starting the sequence, UVM has to be instructed

not to leave the current phase before finishing the current sequence. After the execution finishes, the phase can be advanced. Since sequences can also be structured hierarchically this procedure is only necessary in the top most sequence. To implement this behaviour, the callback functions `pre_body` and `post_body` can be utilised. Listing 7.13 shows how such a synchronization could be implemented.

```
1    void pre_body()
2    {
3        srInfo(this->get_name().c_str())("calling pre_body()");
4        //raise objection if started as root sequence
5        if (this->starting_phase != NULL) {
6            this->starting_phase->raise_objection(this);
7            srInfo(this->get_name().c_str())("objection raised.");
8        }
9    }
```

Listing 7.13: Synchronization of UVM runtime phases

Only when a sequence is started as top most sequence, contains its object variable `starting_phase` a value different from `NULL`. This value specifies the phase in which the sequence was started. If a start phase exists, the `raise_objection` function belonging to it is called. The parameter of the function points to the object causing the objection. This way UVM registers for which components objections exist in what phases (compare also Section 10.1.1) and prohibits premature execution of the next phase. In the callback function `post_body` the objection can be dropped again. This happens by calling the method `drop_objection` of the same phase object that was saved in the `starting_phase` pointer. When no objections exist any more for a phase UVM advances to the next phase.

7.1.8 The *scoreboard*

The scoreboard as a component is completely independent from the other described components. This is achieved through exclusive usage of analysis ports. The scoreboard has ports of the class `uvm_analysis_port` to connect to the other test components. The scoreboard has as many export interfaces as there are components in the hierarchy that provide transaction on their analysis ports. In our example we have two components, one transmitting transactions and one receiving transactions. Hence the two exports are called `xmt_listener_imp` and `rcv_listener_imp`.

The main task of the scoreboard is comparing incoming and outgoing transactions from the DUT. Since these transactions are independent from each other and occur at different points in time, the scoreboard needs to store them. For this purpose the scoreboard contains a FIFO buffer. The UVM SystemC library already comes with a separate class called `uvm_tlm_fifo`. Apart from the default SystemC FIFO class `tlm_fifo` there is also a special class for analysis ports called `tlm_analysis_fifo`, which we will use in our scoreboard.

Listing 7.14 shows how the connections of the exports to the FIFOs are created within the `connect_phase`. Since the analysis export come from the UVM library, but were adapted to use the common method names from System-Verilog, the `connect` function is used instead of the `bind` function.

```
1     void connect_phase(uvm::uvm_phase& phase)
2     {
3         srInfo()("Connect phase");
4
5         xmt_listener_imp.connect(fifo1);
6         rcv_listener_imp.connect(fifo2);
7     }
```

Listing 7.14: Connecting ports in the scoreboard

The `run_phase` shown in Listing 7.15 describes the execution of the scoreboard. It shall compare transactions as long as new ones are received on the analysis ports. Hence the `run_phase` can be modelled as an endless loop. Within this loop transactions are constantly read from the FIFO buffers. The calls to the `get()` functions are blocking and thus it is ensured that two transactions are present when the comparison starts. This behaviour can also be problematic: The monitor and the driver have to be compatible with this behaviour. Assuming a monitor that produces a transaction with each change on the output of the DUT, the FIFO buffers would fill up very fast. It has to be ensured that the driver sends matching transactions to the changes on the output. Only then can be guaranteed that each input transaction has a corresponding output transaction.

```
1     void run_phase(uvm::uvm_phase& phase)
2     {
3         srInfo()("Run phase");
4         while(true)
5         {
6             dut_trans xmt_trans = fifo1.get();
7             dut_trans rcv_trans = fifo2.get();
8             srInfo()
9                 ("transmitted", xmt_trans.convert2string())
10                ("received",rcv_trans.convert2string())
11                ("Comparing transactions:");
12            if(!xmt_trans.compare(rcv_trans))
13            {
14                UVM_ERROR(name(), "Compared transactions are not the same."
                      );
15            }
16            else
17            {
18                UVM_INFO(name(), "Success: Compared transactions are the
                      same.", uvm::UVM_MEDIUM);
19            }
20        }
21    }
```

Listing 7.15: UVM `run_phase` of the scoreboard

Apart from some user output using the usual macros the collected transactions are compared. For the comparison the already mentioned `do_compare` callback function of the transaction class is utilised by calling `compare`. Depending on the operation type of the transaction a corresponding comparison algorithm is selected. The above mentioned macros for UVM messages are used to record success or failure of the comparison. Thereby they are also summarised at the end with the other statistics.

7.2 Integrating UVM components with snapshot manager

All components mentioned above form the UVM test environment. The `sc_main` can be compiled into an executable and the test will run through its sequence and display statistics at the end.

Instead of having to manually configure register values or other internal state variables for a test setup from the sequence it would be desirable to load the required state and directly start the test.

This is possible by integrating the snapshot manager from our snapshotting framework into the test environment. Simply instantiating the `snapshot_manager` module inside the `sc_main` function is sufficient to be able to use its features. Without knowing how the module hierarchy of the test looks, it would be difficult to write a JSON snapshot file from scratch, with which the simulation could be loaded. Therefore, it makes sense to let the whole test run once and save a snapshot at the end to create a baseline snapshot file, which can then be modified. Creating the baseline snapshot is achieved through adding the code shown in Listing 7.16 at the end of the `sc_main` function.

```
1    sm.save_state();
```

Listing 7.16: Creating a baseline snapshot

The snapshot file is rather short and visible in its entirety in Listing 7.17. It shows only registers and signals as other object types have been disabled. The `state_in` signal from the monitor is not listed, because input signals do not store their internal state. The output signals from the driver are numbered and not named as they were created by the factory. We can now modify the `simtime` value and the `counter.register.state` value and tell the simulation to load the snapshot file during the elaboration phase.

Changing internal state registers or other values before running the UVM test, might cause the test to fail. This can be used to also create negative tests or tests with invalid values, that would cause the compiler to complain. The saving and loading of simulation states is controlled through `sr_param` configuration variables. This also creates the possibility to compile a UVM test bench once and run it in a script with easily adaptable inputs or use it in a *continuous integration* setup.

Furthermore, the DUT can be handled as a true black box. The snapshotting framework extracts the necessary information from the model. Enabling register value modification without recompilation usually requires the use of special configuration parameters such as `sr_param`. Using these usually requires having access to the source code of the model under test. With the snapshotting approach also binary DUTs (with accompanying header files) could be tested.

```
{
    "value0": [],
    "simtime": {
        "time": 140000.0
    },
    "value1": {
        "counter.register.state": 0
    },
    "value2": {
        "counter.up": false
    },
    "value3": {
        "counter.reset": false
    },
    "value4": {
        "testbench.agent1.driver.object_6": false
    },
    "value5": {
        "testbench.agent1.driver.object_7": false
    }
}
```

Listing 7.17: JSON snapshot file from UVM platform

7.3 Metrics used in Evaluation

Before the evaluation can be undertaken, the metrics have to be presented. When comparing software frameworks or in the case of our framework software enhancements, there er certain metrics that should be evaluated. In the following, I will give an overview and explain my reasoning for using or not using common metrics.

Overhead When adding new code or a library to a project it is interesting to know how much overhead will be introduced. Overhead can be measured in lines of code for example. While not being an absolute measurement, the lines of code can give an indication about the complexity of a piece of software. Complexity affects compile time the most. When dealing with SystemC simulations, frequent recompilation is common during development and testing.

Performance Reduction Adding more complexity to the code base can also affect overall performance. Performing more tasks certainly will take more time. The performance reduction should not outweigh the functionality gains by the added code. Here we have to evaluate how often certain actions occur. A one time performance hit is fine, but when a function that is executed thousands of times is affected, we have a problem.

Latency The latency by adding checkpointing functionality is closely related to overall performance. During the checkpointing phase, the program is likely in a suspended

state. The latency introduced through the checkpointing feature should be kept to a minimum, especially when doing checkpoints in intervals instead of specific points in time. The latency will add up in the end.

Checkpoint Size Although storage is not a problem in modern workstations, it might become costly, when working on other peoples computers (e.g. *the cloud*). This is also related to overall performance. When the program has to write several Gigabytes of checkpoints, the IO performance of the whole system will be affected.

7.4 The Setup

All tests and evaluations were done on a MacBook Pro with the following specifications: Intel Core i7-4578U clocked at 3 GHz, 16 GB DDR3 memory, 1 TB Apple SSD. As the evaluated software runs better under Linux, everything was executed inside a Ubuntu 14.04.5 VM that has access to 3 CPU cores and 4 GB ram.

A branch for the SoCRocket core repository was created on August 10th 2016 named `bfarkas-devel`. Any upstream code changes to the core components are not considered here. The components `sr_register`, `sr_signal` and `usi` were taken in their latest versions.

For comparison, the user-level checkpointing tool DMTCP was used in version 2.5.2 [136]. It was installed directly from the Git repository. During compilation debug was enabled. Measurements were taken with both executables with and without debugging info enabled.

A stripped down version of the SrCount test platform was created for the DMTCP measurements. Even USI had to be removed, otherwise the checkpointing would not work. This is already one drawback of using such a complex piece of software as DMTCP. It is very difficult to debug issues. It was not clear if the problem was on the SoCRocket side or if DMTCP is not capable of handling executables that include a python interpreter. Although, it is possible to checkpoint Python applications directly with DMTCP and that feature is widely used if the GitHub issue discussions are taken as an indication.

Furthermore, a SrSequencer model was added to the platform to perform the task that the sequencer in the UVM test bench is doing. Since taking checkpoints is not possible from within the SystemC simulation with DMTCP without code modification, the SrSequencer model is configured to provide stimuli in an endless loop.

For the SoCRocket measurements with UVM and snapshotting the platform that was created in Section 7.2 is used.

7.5 Overhead Measurements

The lines of code of a project give a rough estimation about code complexity. This is especially true if more than one programming language is used in the same project. There are a number of tools and scripts that count and analyse lines of code. The tool `cloc` [137] is the most suitable as it has support for many languages and is very reliable.

The UVM and snapshotting framework introduced with this thesis can be split up in three parts: The snapshot manager class (`sr_snapshot`), the extensions to the Cereal library and the Cereal serialization library itself. Tables 7.1, 7.2 and 7.4 show the numbers produced by `cloc` on the respective source directories. Everything is contained in C++ headers and just the Cereal library is rather large with 17 thousand lines of code spread over 79 files.

Language	files	blank	comment	code
C/C++ Header	2	29	51	250

Table 7.1: Output of `cloc` for `sr_snapshot`

Language	files	blank	comment	code
C/C++ Header	4	12	9	88

Table 7.2: Output of `cloc` for `cereal_extensions`

The DMTCP code base is a bit larger. Here `cloc` only analysed the `src` folder inside the project repository. Table 7.5 shows the analysis results. A mix of C/C++ and Assembly and some shell scripts comes in at 34 thousand lines of code spread over 158 files looks already much more complex.

The lines of code only give a rough idea about the complexity and overhead of adding a tool or framework. It might even lead to false assumptions. For this reason, I also measured how long it takes to compile projects that include UVM, the snapshot manager, both or just DMTCP. To get comparable compilation time numbers for SoCRocket platform first the code base was cleaned using `./waf clean` followed by building the desired platform with `./waf build --target=<platformname>`. These two steps were repeated 10 times and the results were averaged. For DMTCP the procedure was similar, the only difference was that `make` was used instead of `waf` as build tool.

Earlier, I have only talked about the lines of code of the add-ons. To better judge the differences in compilation times, it is also worth to have a quick analysis of the SoCRocket core codebase. No platform would be executable without the set of basic building blocks that have been described in Section 3.3.1, namely `sr_register`, `sr_report`, `sr_signal` and USI. Table 7.6 shows the feature sets for the platforms used in this evaluation. The `counter_base` platform is not compiled with USI, as that would break DMTCP functionality. For DMTCP with USI support both frameworks would need to be extended. The `counter_uvm` platform is missing USI support as well, but here it is because it is just a very basic integration test for UVM and the test bench components. The other selected features are self explanatory.

It is clearly visible that all platform rely on several base components. Table 7.7 breaks down the lines of code by component. This should give an idea about the complexity of the base simulation framework. Table 7.8 gives an overview of the languages used in the same base components. The build tooling, several support script as well as parts of USI are written in Python which explains the high percentage of Python code in the components.

Language	files	blank	comment	code
C/C++ Header	11	120	29	569

Table 7.3: Output of `cloc` for UVM test bench

Language	files	blank	comment	code
C/C++ Header	79	4082	7263	17137

Table 7.4: Output of `cloc` for `cereal`

Language	files	blank	comment	code
C++	79	4024	5138	23081
C/C++ Header	61	1203	1655	5288
C	8	624	915	3376
make	5	375	489	2649
Assembly	4	40	115	120
Bourne Shell	1	2	0	11
SUM:	158	6268	8312	34525

Table 7.5: Output of `cloc` for `dmtcp`

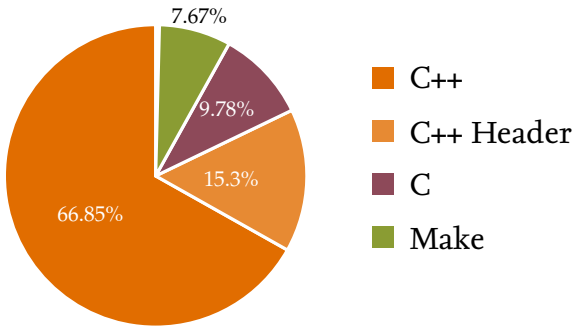

Figure 7.2: Graphical representation of `cloc` output for DMTCP

Feature/Component	`counter_base`	`counter_uvm_sm`	`counter_sm`	`counter_uvm`
UVM		✓		✓
snapshotting		✓	✓	
`sr_register`	✓	✓	✓	✓
`sr_report`	✓	✓	✓	✓
`sr_signal`	✓	✓	✓	✓
USI		✓	✓	
`SrCount`	✓	✓	✓	✓
`SrSequencer`	✓		✓	

Table 7.6: SoCRocket evaluation platform feature matrix

Component	blank	comment	code
common	1243	2525	7902
usi	1105	1056	4860
waf	871	943	4339
sr_param	564	895	2528
sr_register	274	251	1167
sr_signal	164	461	606
sr_report	80	72	423
sr_registry	37	96	233
SUM:	4338	6299	22058

Table 7.7: Output of `cloc` for SoCRocket base components sorted by component

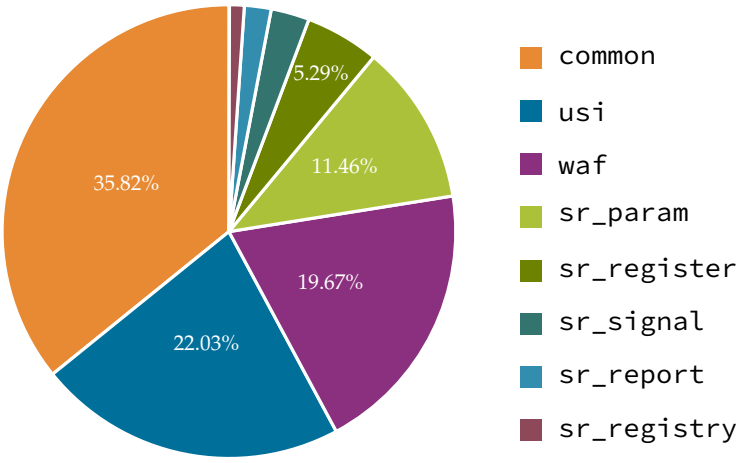

Figure 7.3: Percentages of SoCRocket base components of codebase

Language	files	blank	comment	code
C/C++ Header	94	2270	3945	11117
Python	79	971	1140	4752
C++	29	700	911	4163
CSS	4	281	108	1543
Javascript	5	63	166	305
make	3	50	12	102
HTML	2	1	17	62
Bourne Shell	3	2	0	12
vim script	1	0	0	2
SUM:	220	4338	6299	22058

Table 7.8: Output of `cloc` for SoCRocket base components by language

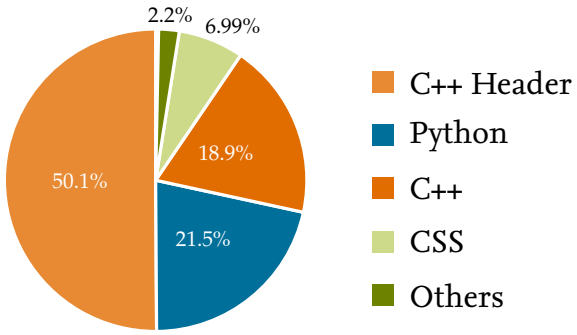

Figure 7.4: Graphical representation of `cloc` output for SoCRocket base components

7.6 Performance and Latency Measurements

The performance and latency criteria are so closely related, that it would not make sense to split them into two different Sections. During development, the task of recompiling code after some changes occurs very frequently. When complex software needs to be compiled this can take quite some time. After every little change, the software or parts of it need to be recompiled. Depending on the software structure, this recompilation step can also take a lot of time if complex headers with lots of macros are included that need to be re-evaluated by the compiler.

The previous Chapter described in detail how my framework is constructed. My framework is compared here against DMTCP, which has not been described in such detail yet.

The distributed checkpointing procedure is coordinated by a checkpoint-coordinator, which handles communication at the barriers. Each application that is using the DMTCP library loads MTCP and executes its setup routine. After setup is done, the checkpoint manager thread can be started and DMTCP opens a TCP/IP connection to the checkpoint coordinator. Saving a checkpoint involves seven steps and six global barriers: 1. Wait for checkpoint request from checkpoint manager thread. 2. While MTCP suspends all user threads, DMTCP is saving the owner of every file descriptor. Afterwards DMTCP waits until all processes have reached this barrier. 3. DMTCP chooses one file descriptor per thread and waits for other nodes to finish. 4. For every socket the previously chosen file descriptor drains the kernel buffers. 5. MTCP writes user process memory to disk. 6. DMTCP sends back socket buffer data to original sender which in turn refills its kernel buffers. 7. MTCP resumes user threads. Restoring a checkpoint works in a similar way. The DMTCP restore process includes a discovery service that is used to identify the new addresses of processes that need to be restored.

DMTCP is a bit different here, as it is not strictly a framework but rather a stand-alone application. For these measurements DMTCP was used as is, without any modifications or special plug-in code to better support SystemC simulations.

To get a baseline we need to compile all the different platform configurations listed in Table 7.6 as well as DMTCP. The time for DMTCP itself is not very interesting, since it needs also a platform to checkpoint to be comparable to the SoCRocket platforms that support snapshotting. The compilation time for DMTCP is summed up with the compilation time for the `counter_base` platform configuration. The compilation times were measured with

the shell built-in function `time`. Figure 7.5 shows the results. The compilation times for the different SoCRocket platforms only vary slightly. It can be seen that adding the snapshotting framework has a higher impact than just adding the UVM framework. This is due to the fact that the snapshotting framework includes the whole Cereal serialization library. Since the library consists purely of C++ headers, the compiler needs much more time to pre-process the code base. For the size of its codebase, DMTCP compiles rather quickly.

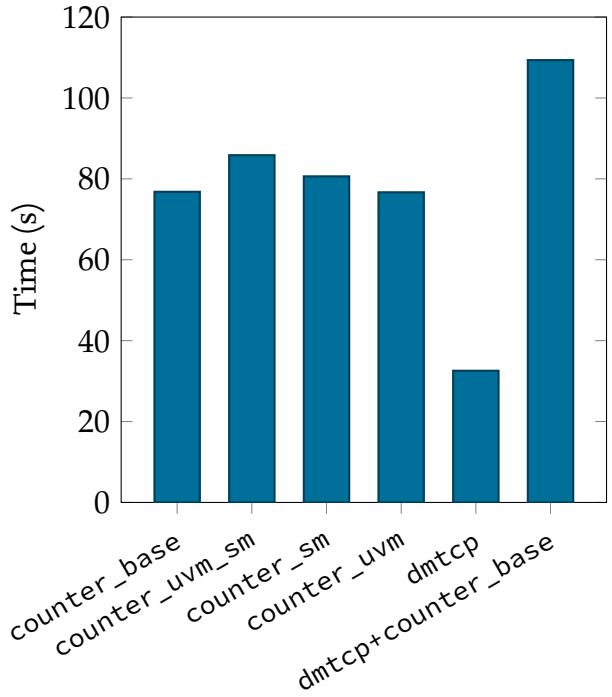

Figure 7.5: Project compile time comparison

It is worth noting here that the `counter_base` platform has almost the same compilation time as the `counter_uvm` platform. Looking at the raw numbers shows that the former even takes slightly longer to compile. This can be explained through the lack of USI in the `counter_base` platform, which had to be removed to achieve compatibility with DMTCP. In the default platforms USI takes care of message logging. When USI is removed the `sr_report` library falls back to a logging implementation involving a lot of C++ templating code, which leads in turn to surprisingly long compilation times.

First time compilation is one thing, but it usually only happens at the beginning of a project or after having to clean the compilation artefacts. The more important case when it comes to working efficiently is the recompilation time. The time it takes to rebuild an executable binary after some code changes. Here, DMTCP was excluded from the comparison since it only needs to be compiled once and its code is not affected by changes in the SoCRocket platform it shall checkpoint. The times were again measured with the shell built-in `time`. Figure 7.6 shows the results. Compared to the basic `counter_base` platform the full featured `counter_uvm_sm` platform increases the recompilation time almost threefold. For these measurements only the `sc_main` function was touched to trigger the recompilation of the platform.

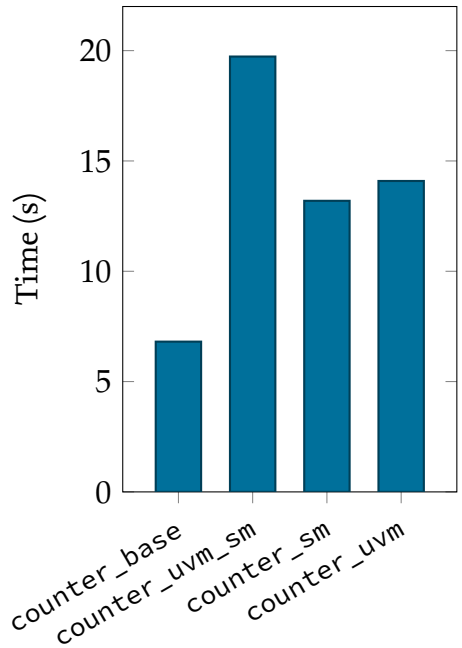

Figure 7.6: `sc_main` recompilation time comparison

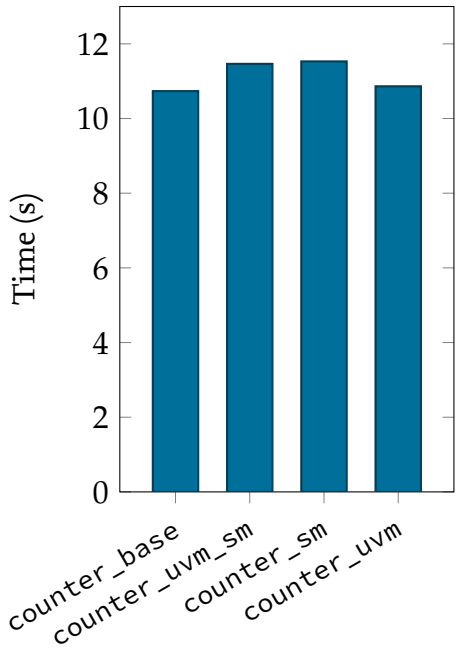

Figure 7.7: SrCount recompilation time comparison

After the platform configuration is established the `sc_main` function usually is not modified very often. Mostly it will be modified when model configuration parameters need to be modified, that cannot be modified through other means such as `sr_param` or USI. It is more likely that during testing or development the model code itself needs to be modified. This can be measured just like modifying the `sc_main` function. The resulting recompilation times for modification of the SrCount model are shown in Figure 7.7. The recompilation times of the SrCount model are much closer together than the times for the `sc_main` function. This can be explained with the model being a bit more independent from the snapshotting and UVM framework than the `sc_main` function.

The only platforms where the recompilation of SrCount takes a bit longer are the ones including the snapshotting framework. This is explained through dependencies towards the snapshotting framework within the model. The serialization library needs to know about the relationship between the base class `sc_core::sc_model` and the derivatives.

Assuming a typical development cycle involves initial compilation, several adjustments to the `sc_main` function and multiple changes in the DUT, we can get an idea how each setup affects overall performance. The numbers of adjustments and changes needed can only be chosen arbitrarily, since they rely heavily on experience.

Four adjustments of `sc_main` and eight changes in SrCount have been chosen for this evaluation. The numbers shown in Figure 7.8 are based on the previous measurements. It looks like the combination of the `counter_base` platform with DMTCP can save some time, but it has to be kept in mind that this combination does not support USI and therefore dynamic reconfiguration of simulation and model parameters is very limited. This might lead to more recompilation cycles and then the initial time saving vanishes. Still, the combination of UVM with the snapshotting framework has the highest overhead due to its

inclusion of many C++ headers.

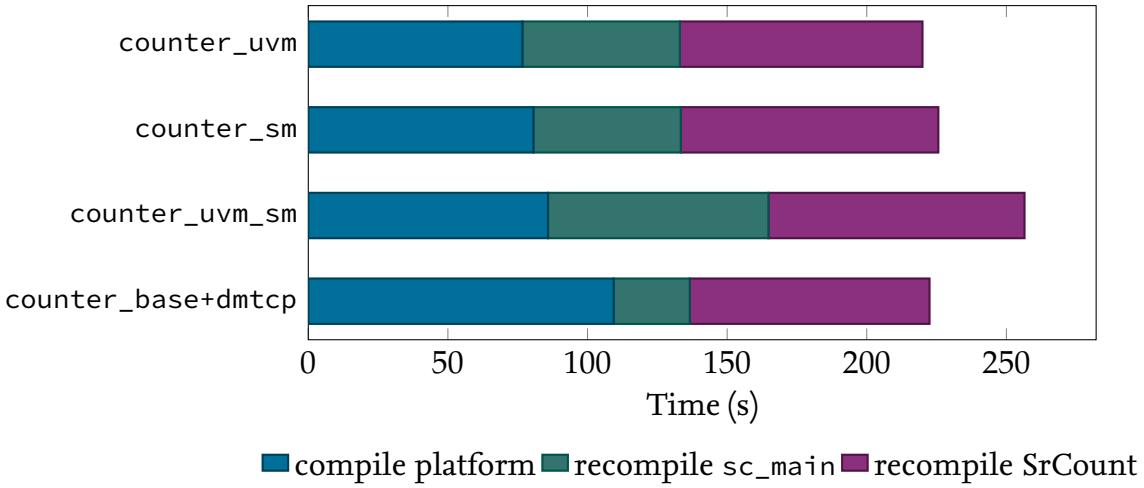

Figure 7.8: Combined compilation and recompilation time comparison

Until now the focus has been solely on compilation times, but not on execution times. When comparing snapshotting frameworks and tools it is also important how long it takes to save a snapshot. During the snapshotting procedure, the simulation or application being snapshotted is suspended. This suspension introduces an execution latency that is directly proportional to the snapshot saving time. Here DMTCP is at a disadvantage, because it has to communicate via external APIs with our SoCRocket platform. Through extra plug-ins for DMTCP and further modifications in the SoCRocket simulation framework it would theoretically be possible to have the simulation talk directly to the DMTCP controller. The author of [116] has integrated MTCP the core component of DMTCP into the SystemC kernel. The framework presented here, works without the need to have a customized SystemC kernel.

The latency of the checkpointing process is defined as the time it takes to halt the simulation, save the full snapshot to disk and then resume the simulation. In SoCRocket the time to save the full snapshot is measured directly in the `sc_main` function using the `std::chrono` library. The corresponding code is shown in Listing 7.18. Measuring the time to halt the simulation and the resuming is a bit more tricky.

```
1    auto t1 = high_resolution_clock::now();
2    sm.save_state();
3    auto t2 = high_resolution_clock::now();
4
5    duration<double, std::milli> duration_ms = t2 - t1;
6    cout << "### savetime = " << duration_ms.count() << " ms" << endl;
```

Listing 7.18: Measuring time inside `sc_main`

Resuming the simulation is realized through simulation phase callbacks, so the time spent in each callback function has to be measured. Halting the simulation is done as part of the save process, so the time measurement had to be included inside the `save_state`

function. These times will then be added up to get a number for the latency that can be compared against DMTCP.

The pie chart in Figure 7.9 shows the percentage breakdown of the snapshot latency time for my snapshotting framework. Each piece represents one function that is necessary for saving and restoring snapshots. The time to halt the simulation is so small that it does not show up in the chart.

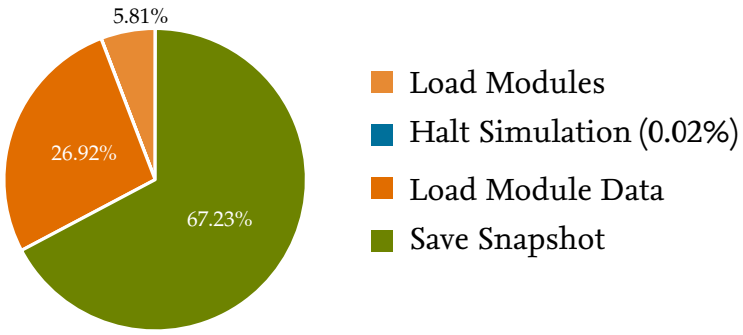

Figure 7.9: Snapshot time breakdown by functions

The DMTCP binaries were not touched to not interfere with their operation. Instead, the shell built-in `time` function was used again. The command `dmtcp_command -bc` was used to take a snapshot and measure the time. The parameter `-bc` causes the program to block until the checkpoint is completed. Taking a snapshot with DMTCP involves three processes: The SoCRocket simulation platform (started with `dmtcp_launch`), the `dmtcp_coordinator` and the aforementioned `dmtcp_command` tool to interact with the coordinator. Figure 7.10 gives an overview how these processes are connected. The sequence diagram in Figure 7.11 shows the sequence for acquiring the checkpointing time with DMTCP using two shell sessions.

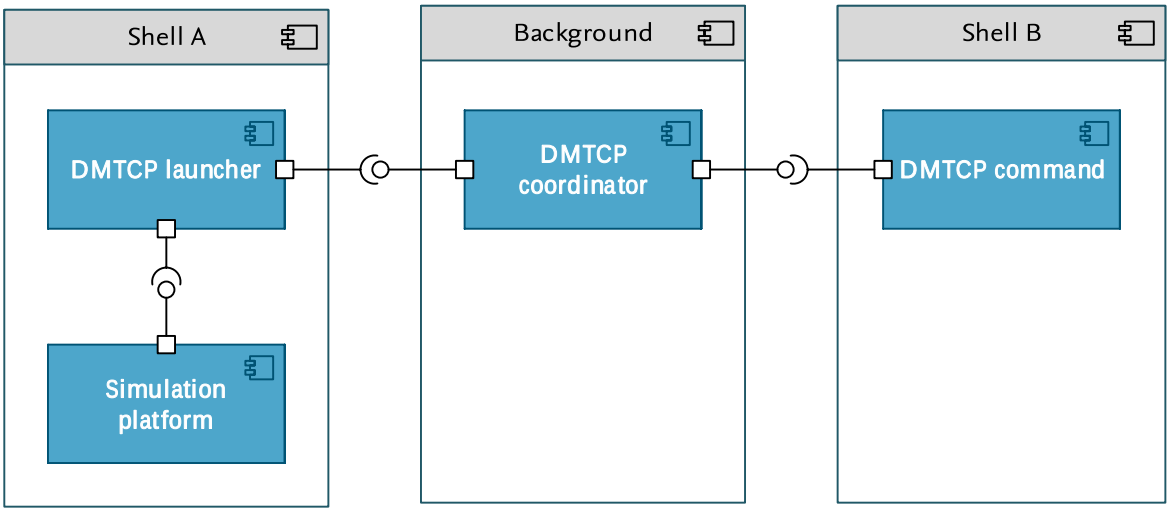

Figure 7.10: Component diagram for processes involved in DMTCP time measurements

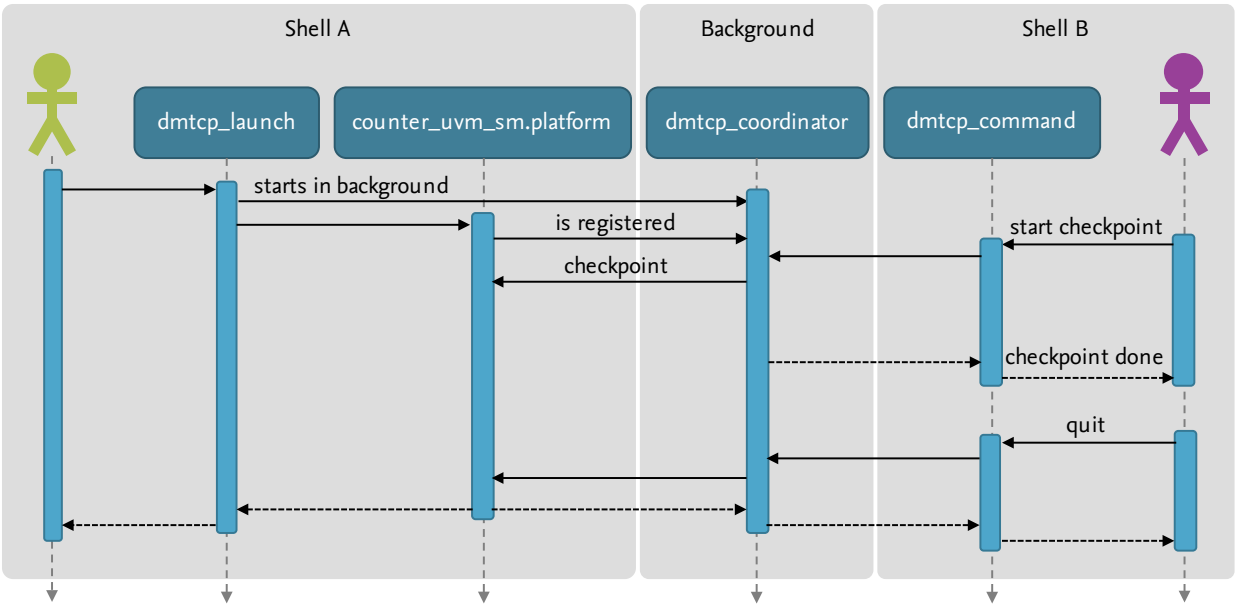

Figure 7.11: DMTCP checkpointing processes

The `dmtcp_launch` tool instruments the binary so that it can register itself with the coordinator and can be controlled from it. Initially DMTCP and its supporting tools were built with debug information enabled. So the checkpoint time measurements were done with both binaries. The results are shown in Figure 7.12. The times are averaged over 10 measurements. The chart is using a logarithmic time axis so that the time for `sr_snapshot becomes visible`. Using debug binaries or not does not make much of difference for DMTCP. The difference between `sr_snapshot` and DMTCP however is four orders of magnitude. Clearly the lightweight integrated solution integrated directly with the simulation binary is performing much better here. For occasional snapshots of almost any application DMTCP is good enough. It does not offer much flexibility but is quite user friendly.

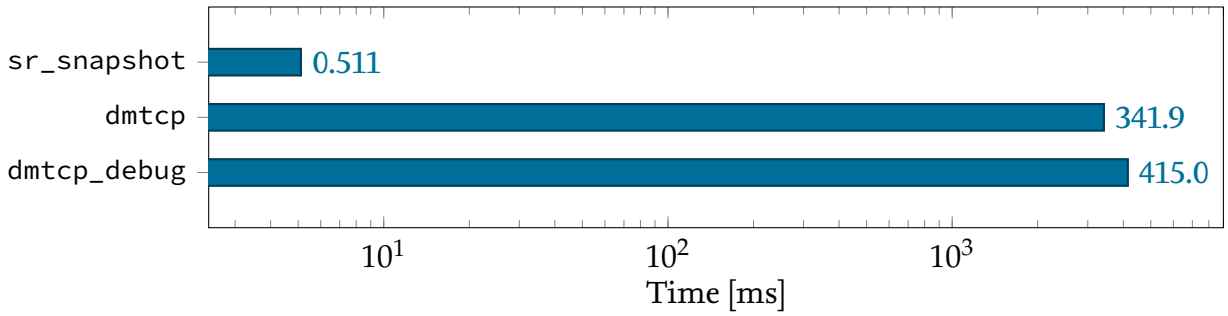

Figure 7.12: Snapshot time comparison

7.7 Checkpoint Size Measurements

The last metric to be evaluated is checkpoint size. The measurement procedure is very straightforward here. After successfully creating several snapshots during the previous time

measurements, the file sizes of the resulting checkpoint files are compared with the `ls` tool. Figure 7.13 shows the file sizes in bytes. Again a logarithmic scale had to be used as the checkpoint files from DMTCP were several orders of magnitude larger than the ones from `sr_snapshot`. The checkpoint files of DMTCP contain not only internal state data, but the whole process memory and necessary meta-data to reconstruct the checkpointed process with its threads as it was at the time during the snapshot. All this information is stored in a binary format with optional compression. Whereas the checkpoint file of `sr_snapshot` is a concise and human-readable JSON file containing just enough information to reconstruct the simulation when it is restarted. The JSON format adds the ability to also edit checkpoints and thereby modify internal states of the simulation models contained in the platform. Furthermore, JSON snapshots offer portability, while it is unclear if a DMTCP checkpoint can be used on a different machine than the one it was created on.

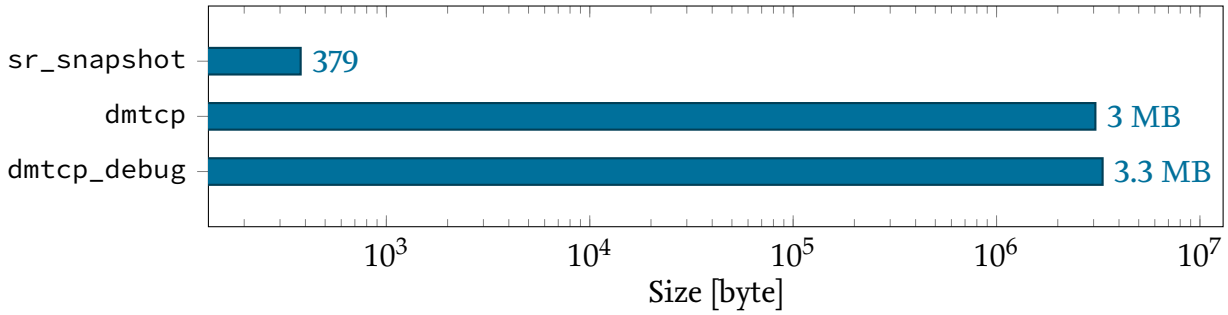

Figure 7.13: Snapshot size comparison

7.8 Gaisler IRQMP Evaluation

In this thesis, I am working with one specific IP core from Cobham Gaisler's GRLIB, as their IP cores represent the base for the SoCRocket SystemC models. GRLIB contains several VHDL IP cores for developing SoCs and is published as free software under the GNU General Public License (GPL) at [7]. The centrepiece of the GRLIB is the LEON3 Processor IP core. The LEON3 is a 32-bit processor using the SPARC V8 architecture. Nearly all other IP cores are connected to the LEON3 per the Advanced Microcontroller Bus Architecture (AMBA) 2.0 bus, specifically AHB and APB.

7.8.1 Multiprocessor Interrupt Controller

Multiprocessor Interrupt Controller (IRQMP) has been selected as Device under Test (DUT) for the checkpointing UVM use-case. It is a fairly straightforward component with well-defined inputs and outputs as well as an internal state. This component will be introduced briefly for easier understanding of the use case described in this Section.

As can be seen in Figure 7.14 the IRQMP component has just a few interfaces. The IRQMP component is responsible for distributing IRQs to a system's processors, supporting up to 16 processors at the same time. The interrupt system is laid out as an interrupt bus in parallel to the AMBA bus, with the distinction that this bus is implemented using the *sr_signal* library.

IRQMP itself is connected to the APB bus as slave and surveys the combined interrupt signals. It processes the incoming interrupts by prioritizing. Furthermore, certain interrupts

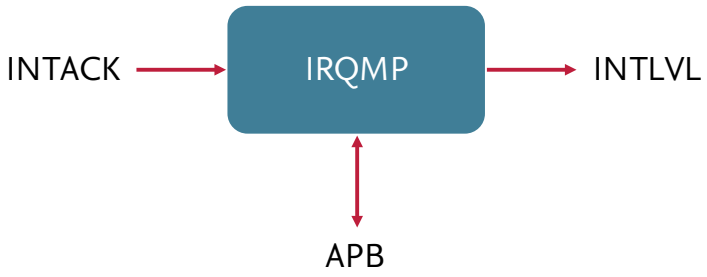

Figure 7.14: IRQMP and its interfaces

can be disabled using a masking mechanism. After IRQMP has processed pending interrupts, it enables the output signal (INTLVL in Figure 7.14) for the remaining interrupt with the highest priority and thereby informs the connected processors. Figure 7.15 shows a general configuration for using IRQMP in a SoC.

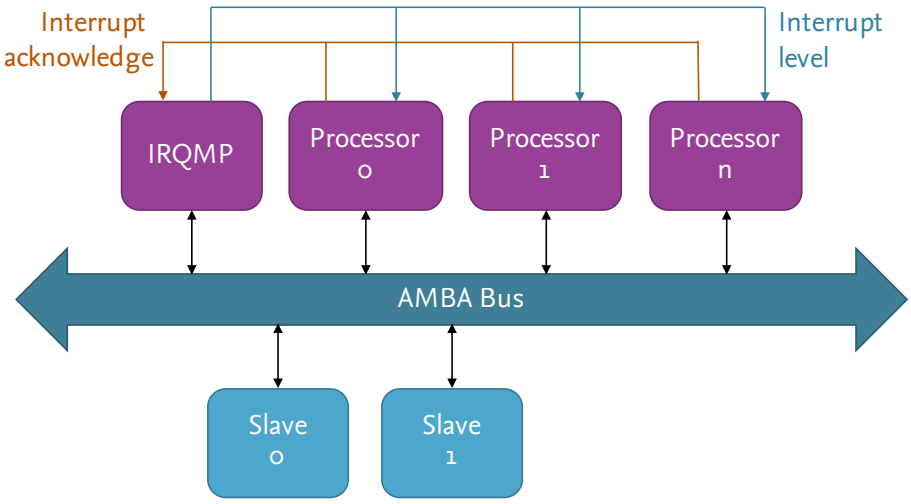

Figure 7.15: IRQMP topology adapted from [138]

IRQMP has a 32 Bit wide input `irq_in` implemented as `infield` (see 6.1.2). During normal operation, only Bits 1 through 15 are supervised. Bit 0 is reserved. If one of the supervised Bits is set, the corresponding Bit is set within the *interrupt pending register*. Even after the state of the incoming signal wire has changed, the value inside the register stays the same. The register value can only be changed by software or by the corresponding processor's *interrupt-acknowledge* signal (see Figure 7.15) of the corresponding processor. Each of the 15 interrupts can be assigned one of two interrupt priority levels, 0 or 1, with 1 taking precedence over 0. Within these two levels, the interrupts are implicitly prioritized with higher interrupt numbers reflecting higher priority.

In case one interrupt is not allowed to reach a certain processor, the masking mechanism can be used. For this, IRQMP has unique registers for each processor, i.e. *processor n interrupt mask register*, with *n* being the processor index. In contrast to the prioritization, the masking of interrupts takes effect for each single processor.

The *interrupt force register* can be used to set a specific interrupt directly. Again, one register is allocated for each processor. If a processor acknowledges an interrupt set through its

interrupt force register using the matching *interrupt acknowledge* signal, the corresponding Bit in the *force register* is reset and not the one in the *pending register*. This mechanism is especially useful for the implementation of interrupt broadcasting. If one interrupt reaches multiple processors, the *interrupt acknowledge* signal of the processor which answers first resets the interrupt Bit in the *pending register*. Should all processors acknowledge the interrupt, the broadcasting mechanism will set the corresponding bits in the processors *force registers*. This mechanism is configured in the *broadcasting register*. This way ensures that each processor by itself has to acknowledge an interrupt of this kind.

Additionally, IRQMP supports an extended mode to allow using the upper 15 Bits of the interrupt bus. The extended mode is enabled by providing the parameter `eirq` during instantiation. The parameter specifies which normal interrupt is triggered when one of the extended interrupts is set. If a processor acknowledges an extended interrupt, both *pending bits*, of the normal and of the extended, interrupt are reset. For this to work, IRQMP saves each triggered extended interrupt in an additional register. If said register has a value other than 0 for the interrupt specified with `eirq` while the interrupt is acknowledged by a processor it is clear that a normal interrupt was triggered by an extended interrupt. If the opposite is the case, a normal interrupt was certainly triggered as such.

Apart from the here described features, IRQMP can be used to monitor, pause, and resume processors. For understanding the implemented use case, the basic features are sufficient. Further information can be gained from Gaisler's IRQMP documentation [138].

7.8.2 Integration of IRQMP with snapshotting framework

To demonstrate a real-life use case for snapshotting within a test framework, the IRQMP component is used as DUT within a UVM test bench. The test bench developed earlier for the simple counter platform could largely be reused. Only the driver and monitor were adapted to fit with the IRQMP interfaces. Of course also the test sequence was adapted to fit the use case of exercising an IRQMP instead of a simple counter state machine. This shows already how versatile UVM is and how usable the early version of its SystemC/TLM implementation is.

The snapshot manager class can be used as is. IRQMP only relies on core functionality of the SoCRocket framework, which is already supported with the implemented Cereal extensions. As mentioned above, IRQMP uses *sr_signal* signals for IRQ communication as well as *sr_register* registers for its register implementation. Bus communication is handled through existing classes.

Figure 7.16 shows the SystemC simulation structure used to integrate IRQMP with UVM and snapshotting. The snapshotting manager class is not shown here, as it is doing its work in the background and not playing an active part in this simulation. The `sc_main` contains the instantiation of IRQMP and the test bench. The internal structure of the test bench is defined by its config and instantiated through the UVM factory. The test bench comprises two agents and one scoreboard. One agent is configured to act as a driver and plays back the sequence. A part of the test sequence used to instrument IRQMP via the APB bus can be seen in Listing 7.19. It shows the needed bus transactions for checking the value of an internal IRQMP register. The second agent is configured as a monitor and watches the `irq_req` output of IRQMP. The scoreboard contains listeners for both agents and can check if the output for each input is as expected. Once the sequence is finished the scoreboard will print an overview of the test results.

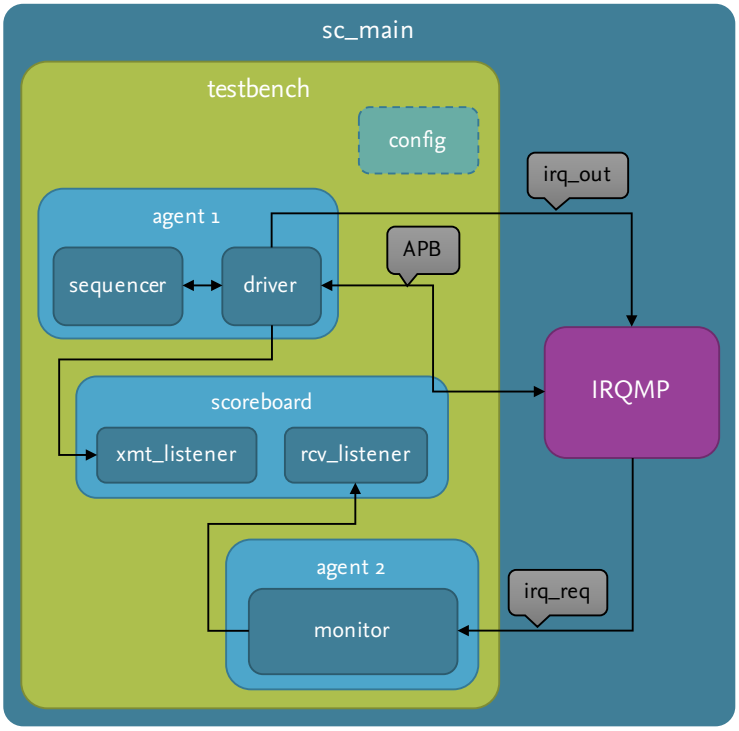

Figure 7.16: UVM test bench configuration for IRQMP

```
////////////////////////////////
// Check IRQ CPU 1 mask reg //
////////////////////////////////

req = new REQ();
rsp = new RSP();
req->addr = 0x00  /* Base address */ + 0x40 + 4 * 1  /* irq cpu 1 mask reg
   */;
req->data = (1 <<  3) | (1 <<  4) | (1 <<  7) | (1 <<  8) | (1 << 11) |
            (1 << 12) | (1 << 15) | (1 << 16) | (1 << 19) | (1 << 20) |
            (1 << 23) | (1 << 24) | (1 << 27) | (1 << 28) | (1 << 31);
req->op = CONFIG_READ;

v::info << get_full_name() << "Check IRQ CPU 1 mask reg" <<
        v::endl;

this->start_item(req);
this->finish_item(req);
this->get_response(rsp);

if(rsp->data != req->data)
{
    std::ostringstream str;
```

```
23      str << "Error, address: 0x" << std::hex << req->addr;
24      str << " expected data: 0x" << std::hex << req->data;
25      str << " actual data: 0x" << std::hex << rsp->data << std::endl;
26      UVM_ERROR(this->get_name(), str.str());
27  }
28  else
29  {
30      std::ostringstream str;
31      str << "Success, address: 0x" << std::hex << req->addr;
32      str << " expected data: 0x" << std::hex << req->data;
33      str << " actual data: 0x" << std::hex << rsp->data << std::endl;
34      UVM_INFO(this->get_name(), str.str(), uvm::UVM_MEDIUM);
35  }
```

Listing 7.19: Excerpt of UVM sequence for IRQMP test bench

Listing 10.4 in the appendix shows an example output of a snapshot taken after the UVM test bench was finished with exercising the IRQMP component. As can be seen that the IRQMP signals as well as the test bench driver and monitor signals are captured in the snapshot.

```
1  --- UVM Report Summary ---
2
3  ** Report counts by severity
4  UVM_INFO      :  55
5  UVM_WARNING   :   0
6  UVM_ERROR     :   0
7  UVM_FATAL     :   0
8  ** Report counts by id
9  [RNTST]                 1
10 [sequence<REQ, RSP>]    6
11 [testbench.agent1.driver]                39
12 [testbench.agent1.monitor]                 1
13 [testbench.scoreboard0]                  8
```

Listing 7.20: UVM report created by the test bench

In Listing 7.20, an example report output for the IRQMP test bench is presented. Loading the snapshot saved at the end of the tests and starting the tests again yields different results. As the state of IRQMP is not anymore as expected, most of the tests in the sequence will fail. Picking diffirent points in time for the snapshotting can therefore be used to cause some unexpected events that might be difficult to specify in the sequence itself or have the whole IRQMP set in a specific state by simply adapting the snapshot file instead of having to modify the test bench code and recompile.

7.9 Summary

In this Chapter, the presented solution was thoroughly evaluated using a realistic use-case and compared against the competing solution *DMTCP*. It was demonstrated how the presented solution clearly fulfills all requirements as defined in Section 4.6.

First, the framework had to be extended for usage with the UVM library from the previous implementation that focussed on a very basic example platform. As a start the basic counter platform from the previous Chapter was introduced as a DUT in the UVM test bench. A driver and an agent had to be implemented to fit the DUT. Additionally, a simple sequence was created to instrument the DUT in order to record some test results in the scoreboard component. These UVM test bench components later provided a good base to implement the real life use case of testing the *IRQMP* SoCRocket model. As expexted, the UVM test bench integration went straightforward with the snapshot manager class and snapshotting of the whole test bench was possible.

Before moving on to the *IRQMP* example use case, the snapshotting framework had to be evaluated against *DMTCP*. For this evaluation I selected the metrics *overhead*, *performance reduction*, *latency*, and *checkpoint size*. The codebase for *DMTCP* is much more complex than my snapshotting framework including the Cereal serialization library. So for the *overhead* metric my snapshotting framework has a clear advantage. Especially as it is directly integrated into the SystemC simulation. *Performance* and *latency* were evaluated by measuring compilation and execution times. My snapshotting framework was used with several SystemC simulation platform with varying feature sets, whereas *DMTCP* could only be used with one SystemC simulation platform. My application-level snapshotting framework outperformed the user-level solution *DMTCP* by several order of magniture for the snapshotting time. Compilation times were similar with a small overhead added through the addition of the *DMTCP* build process. When comparing recompilation time for the main platform and a single module together with the snapshotting framework, *DMTCP* has a slight advantage since it does not need to be recompiled with each SystemC platform change. The evaluation of the *checkpoint size* was again straightforward. Since *DMTCP* uses a binary format to store the snapshotted application state, my snapshotting solution is a clear winner here, with just a few bytes of textdata needed to store the necessary state data for a simulation. My solution even allows for editing of the snapshot files, which is next to impossible with *DMTCP*.

Lastly, my snapshotting framework together with the UVM library was used to run tests for the *IRQMP* model and take a snapshot of its internal state. The UVM test bench that was implemented for the simple counter platform needed only small adaptions to work with the *IRQMP* model. The driver, monitor, scoreboard and sequence needed to be adapted. The sequence was filled with several tests to instrument the *IRQMP* model. The snapshot manager class did not need any further adaptions, since the *IRQMP* model uses standard SystemC features and the SoCRocket signal and register implementations.

In this evaluation I have shown how my snapshotting framework outperforms an established snapshotting solution in several metrics, such as overhead, performance, latency and checkpoint size. With the integration into a realistic UVM test bench setup, I have further shown that my solution is not only a proof-of-concept implementation, but is already applicable in real use-cases. My snapshotting framework is based on current C++ standards. It is therefore possible to abstract most of the complexity of the snapshotting process away, which in turn enables model developers to focus on their task. They will not have to modify

the snapshot managing code like in previous SystemC snapshotting iterations. With my snapshotting solution they can simply work with their existing models and concentrate on the modeling code instead of framework code.

In the next Chapter, I will discuss my work and show how it fits in with the current SystemC standardization roadmap. Furthermore, I will point out areas where future work can improve on my snapshotting solution.

8 Discussion

In the previous Chapters, I have presented my implementation of a SystemC snapshotting framework. Furthermore, I have shown that my snapshotting framework is applicable for real world use cases. It can for several use cases outshine established open source tools such as DMTCP.

In this Chapter, I will look into the positive aspects of my solution. Moreover, I will also try to depict areas where my snapshotting framework can be improved further.

Determining how my solution fits in with the SystemC standardisation roadmap will follow. Furthermore, I will lay out possible steps to get my approach adopted for inclusion in the reference SystemC kernel implementation. Then I will show how my work fits in with other current work on SystemC simulators and especially the SoCRocket framework and how these works can be combined to create new functionality.

Furthermore, I will try to show how my snapshotting approach could be useful in other application domains apart from hardware design for the space domain.

8.1 The Good, The Bad and The Ugly

With the snapshotting framework presented in this work, a first step towards generic snapshotting within SystemC has been undertaken. The evaluation in the previous Chapter has shown that the implementation works and performs well compared to an established application-level checkpointing solution such as DMTCP. As has been laid out in Section 4.5 other works have implemented SystemC checkpointing with the help of external tools already. Implementing the serialization functionality directly in the simulation framework offers much better performance and flexibility.

The snapshots are saved in plain text JSON format, which makes them very compact. Another benefit from using a text-based format for storing snapshot information is its modifiability. A developer can just open the files in his or her favourite editor, modify a few values and see how the models will react. Furthermore, snapshot files can be versioned easily in a version control system such as Git and thus the history of subsequent snapshots can be traced without much effort.

Having compact and modifiable snapshots makes them also portable. Snapshot files can be shared between developers effortlessly. The only limitation is that a snapshot created with a specific version of the used serialization library has to be restored with the same version. To make this a bit easier, the Cereal serialization library supports versioning of snapshot files. When versioning for snapshot files is used, the compiler will report mismatched library versions.

Integrating the snapshotting functionality directly in the SystemC kernel, would create a fork of the SystemC reference implementation. If vendors want to use their own SystemC implementation they would need to patch it accordingly. My current implementation of the snapshotting framework relies only on SystemC standard APIs and should work with other SystemC kernel implementations. Unfortunately, no other kernel implementations were available for evaluation.

The SoCRocket scripting functionality established with USI has been relatively underused in this thesis. The scripting interface is mainly used for simulation control, logging and configuration. Model configuration variables that have been specified as `sr_param` could be modified in Python scripts or in the interactive Python console. Even register values could have been examined. The snapshot manager class already has the same functionality, so there was no further need to have the same functions available in Python. The goal of my work was after all to show how a SystemC snapshotting framework can be implemented in a way that is compatible with the reference simulator and follows SystemC standards. Scripting interfaces for the SystemC reference simulator are not yet standardized and I did not want to introduce another dependency, apart from the serialization library.

In the proof-of-concept implementation, the save and restore functions were added manually to models and certain data types. When code generation tools are used to create models from register description files or other forms of more abstract descriptions, it is possible to generate the save and restore functions automatically. That would enable a modelling framework and code generator to support snapshotting for any model that is created with it directly out of the box. This improvement I will describe in more detail in the next Section.

The SystemC implementation of UVM can take full advantage of C++ language features. Already the reference implementation uses the factory pattern to create test benches and their subcomponents from a configuration database. This enables developers to create models that can be reused easily through changing their configuration parameters. That is already quite close to code generation, if test bench developers take care to parametrize their models extensively.

The basic UVM test bench components developed within this thesis can already be adjusted without much effort for other DUTs. As has been shown in the previous Chapter, with the adoption of Gaisler's IRQMP as DUT, the process was not automated yet, but the code changes were really minimal.

The approach to UVM test bench setup in this thesis was slightly different than the default setup used with UVM test benches. A test bench usually contains a reference model written in SystemVerilog. This reference model should behave according to the models specification. It would also be possible to use an existing RTL model as reference for the SystemC model under test. This approach would lose the benefits of faster simulation times that SystemC simulations offer. Furthermore, creating proper UVM drivers and monitors that translate transactions into signals for the RTL model is more error prone.

One drawback of the current implementation of the snapshot manager is that right now only the loading of internal state for a model is possible. In future, it should also be possible to load complete modules. This feature would require some more adaptions to the SoCRocket simulation framework, which I will come back to in the next Section. One possible way to realize loading of whole models, including state, could be through using the factory included with UVM SystemC. However, this would again introduce another dependency.

The Accellera CCI working group has save and restore on their roadmap, which is shown in Figure 8.1. When they reach that milestone in their roadmap, they could consider my approach for snapshotting. This way my approach could find its way into the reference SystemC kernel implementation and from there into proprietary implementations.

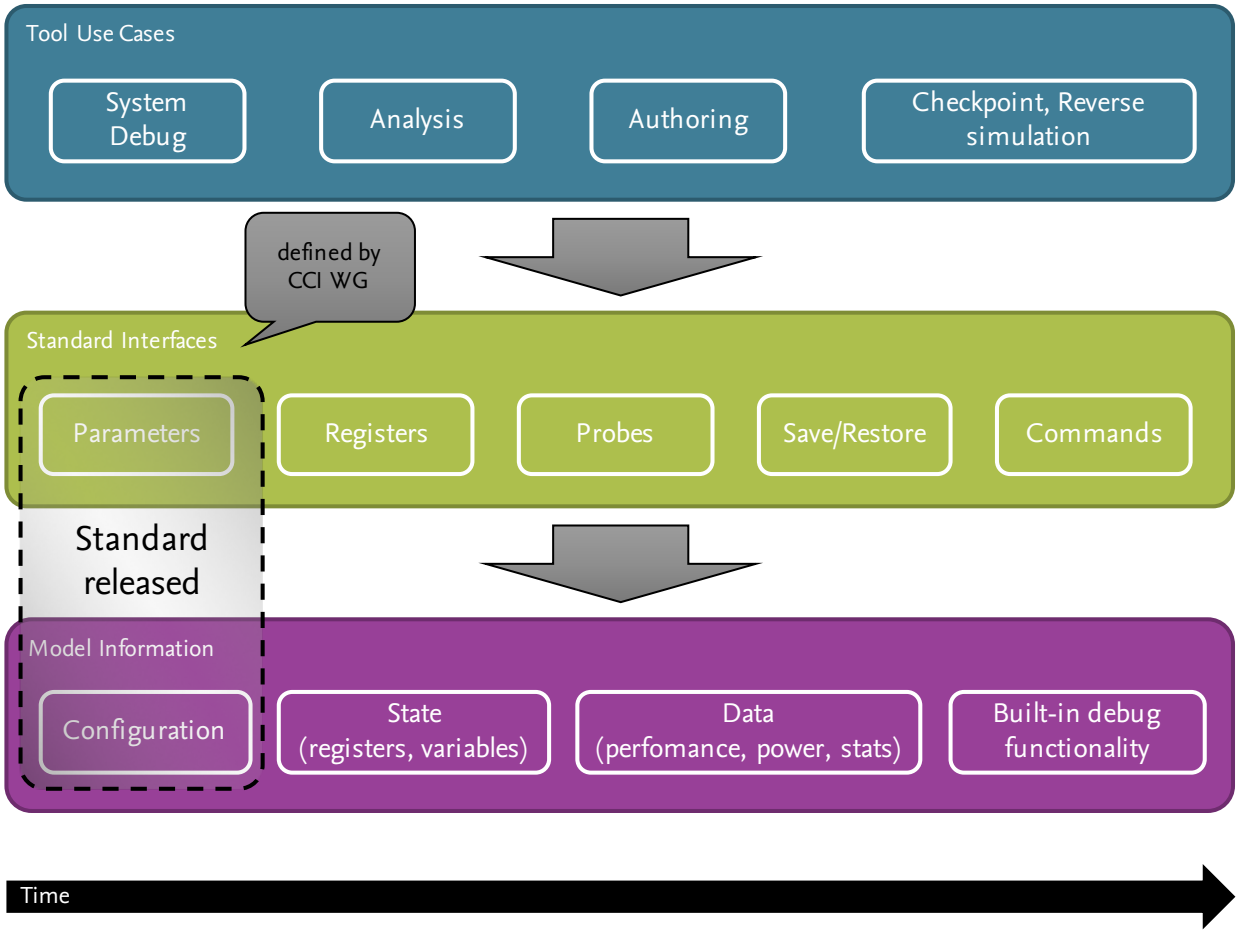

Figure 8.1: CCI working group roadmap adapted from [140]

8.2 The Shape of Things to Come

In the previous Section, I have reflected on the good and improvable aspects of my own work. In this Section, I will try to put my own work into perspective of other current and ongoing works in the area of SystemC simulations. Throughout this Section, I try to forecast where SystemC snapshotting and UVM testing is headed.

The Accellera CCI working group has at this point in time just wrapped up standardisation of configuration parameters. In their roadmap, depicted in Figure 8.1, that is just the first step for interface standardization. Next in line will be registers, followed by probes and after those save and restore interfaces will be their focus. So it might be quite a long way until checkpointing will make its way into the SystemC kernel and reference library. Configuration parameters standardization started already in 2013. The SoCRocket parameters are not yet aligned with the latest standardized configuration parameters from the CCI working group. The official parameters are based on work by Greenhills. Since SoCRocket parameters were originally also derived from Greenhills work, it should be not very difficult to adapt them to the official standard. At DVCon Europe 2017 the CCI working group [139] has released the configuration draft standard for public review [140] and in June 2018 the first version of the standard was officially released [141].

In October 2017 another SystemC Evolution Day was held [142] in Munich. One of the

workshops had the topic checkpointing and SystemC [143]. At that workshop Engblom and Zeffer gave a general overview about snapshotting and presented their proof-of-concept implementation for SystemC checkpointing. Their solution uses the Boost serialization library and relies on the checkpointing functionality of the Simics simulator. Furthermore, they relied on modifications to an Intel-internal SystemC kernel to make their proof-of-concept work. My work presented here can also help move the discussion further. If my approach is adopted into the reference SystemC kernel implementation, some of my workarounds, such as the one in Section 6.2.1, could be neatly integrated directly into the kernel. Once the approach is standardized, the door would be open for proprietary implementations to follow suit and adopt the same approach.

The next step in the CCI working group roadmap after parameters and registers is probes. A first step here could be standardizing an interchange format for recorded transactions. Probes as standard interfaces could be used to enable record and replay of TLM transactions. A feature that would also be of importance to the UVM SystemC library. Right now this feature is already available in extra tooling from some vendors, but not yet widely available or even standardized.

On a side note, the SystemC UVM library has only been tested with loosely timed models. In the future, support for approximately timed models will be added as well.

With standardized model probes, recorded transactions could replace the reference model within a UVM test bench. The reference input and output can be recorded once during a very accurate SystemVerilog RTL simulation and then used to test and verify models written in SystemC. The assumption here is that the playing back of transactions in a simulation is much faster than having to simulate two different levels of abstraction at the same time in lock step.

My current implementation of the *test* inside the UVM test bench relies on comparing input and output transactions within the *scoreboard*. The transactions are transmitted from the *driver* and *monitor* via *analysis ports*. While reporting errors with the `UVM_ERROR` macro, the emitting component is the *scoreboard* itself. The comparison function inside the transaction class is creating a string describing which transactions are compared, but this makes locating errors challenging.

One solution would be to modify the `do_compare` function of the transaction class. The comparison method would still be chosen depending on the operation type, but in addition to just returning a bool value the `UVM_ERROR` macro can be called. This time it would display the name of the affected transaction. Moreover, it would be possible to assign a unique name to each transaction object to make them identifiable. These could then be passed to the macro and discovering error locations would be much simpler.

Alternatively the `do_compare` function could stay untouched and the output in the *scoreboard* could be enhanced by accessing the additional transaction information mentioned above. This would be the preferred method as it gives more flexibility and the additional transaction information is useful in any case. This solution would also keep the test bench code reusable.

Writing test cases and their surrounding test bench code is a lot of manual effort for engineers. This type of repetitive work can benefit largely from automatic code generators. Automatic generation of test code is very convenient when tests are limited to registers. As soon as tests involve functionality instead of just storage the developer has to be involved

again.

Vendors such as Cadence include code generators with their SystemC/TLM tool offerings (e.g. [144]). One such code generator is "tlmgen" [145] which is included in the *Virtual System Platform* offered by Cadence. The tool can generate TLM models from IP-XACT [98] or other textual register description files. Models created by the tool offer no functionality except the register interface. Any real functionality has to be added manually through extension of the created code.

Since many models rely on registers to store their state, it would be helpful if the code generator also automatically generates the necessary code to snapshot the generated models. If developers then extend the models functionality and need to also save some internal variables, it would be very easy for them to just add the additional variables to the existing snapshot functions instead of having to write the snapshot functions from scratch.

Of course automatic code generation only works when creating models from register descriptions. With a large catalogue of existing models there has to be another option to reduce the amount of manual work for a developer to add the necessary snapshotting functionality to existing models. One such option is creating a Python script using the `CppHeaderParser` package [146]. This allows to statically analyse C/C++ header files and gather the introspection information which is not available directly from within C/C++. Information about the class structure and internal variables can then be used to create the necessary functions to snapshot an existing model. This can be done by extending the existing model class and using existing snapshot manager functionality to access private class members. If the headers are available, it is also possible to instrument binary only models in this way.

With advancements in standardization of CCI standard interfaces and further changes in the general structure of the SoCRocket model library, it will be possible to replace the just mentioned *CppHeaderParser* script with a dynamically loadable USI script.

The snapshot manager could generally benefit from tighter integration with USI. Currently, the checkpointing process of the snapshot manager class is controlled by configuration parameters from the command line or inside the `sc_main` function of the simulation. This allows for basic control of the feature. Checkpoints at specific timestamps have to be specified within the `sc_main` function, while command line arguments can be used to switch the whole feature on or off. With better integration into the USI framework, it will be possible to control snapshotting from within Python scripts or the interactive Python console. This enhancement would also allow creating periodic snapshots more easily than currently possible. The question for periodic snapshots is how much data should be kept or if just the deltas to the previous snapshot shall be stored. This could entirely be done within the controlling Python script, if it has direct access to the JSON snapshot from within Python. Python has very good JSON support. Extending USI as well as adapting the snapshot manager class would allow handling the JSON data directly in Python and thereby enabling more flexible processing of the snapshot information.

It is certainly possible to run Python code as callbacks during SystemC simulator phase events. Furthermore, it is possible through hooking into these phases to have Python function called periodically from the SystemC kernel. This would make it possible to listen for specific events and create a snapshot at that point in time or have periodic snapshotting. Of course this would also be possible with the plain C++ snapshot manager, but the scripting

interface offers the ability to achieve new functionality without the need to recompile the simulation code. Recompilation can take quite some time as I have shown in Section 7.6.

There is ongoing work to enable model writing directly in Python. Furthermore, USI will enable dynamic loading of precompiled models, if they were compiled to support dynamic linking.

Writing models in Python is not very interesting for the snapshotting functionality. If the models use the given register and signal implementations, the snapshot manager will be able to save and restore their states. Dynamically loading models is much more intriguing, as it will also enable restoring the models state, when it is dynamically loaded. This requires further attention and work in the future. The aforementioned *CppHeaderParser* Python module could prove to be useful here. Dynamic loading of precompiled models and writing models directly in Python is explored by Meyer in his PhD thesis [147].

It could then be possible to load a model, take a snapshot and then load a different version of the model and restore the snapshot into the new version. This is not strictly related to the scripting interface. As it could also be done non-dynamically by swapping the model code and recompilation. USI simply offers the ability to skip the recompilation step and load precompiled models directly, at least in theory. One requirement here would be strict adherence to the same API. As soon as the API of the model changes, the snapshot will not be usable any more. For a model the API encompasses signals, registers and internal state variables.

Signal, registers and internal state variable are also interesting for fault injection into simulations. There is ongoing research in this area. There are many possible interfaces where this could make sense, although the aforementioned ones are the most likely to suffer from faults in real hardware. Most hardware fault can be traced back to faulty memory or memory errors induced by radiation. While radiation is mainly a problem in the space domain [148], with ever smaller feature sizes in consumer chips, cosmic radiation becomes relevant in other sectors as well [149, 150].

With my snapshotting framework introduced in this thesis, it would be possible to do simple offline fault injection, by simply modifying register and signal values in the snapshot files. With the help of USI and a Python script that can inject faults with statistic probabilities into register and signal values, it is possible to find out how a model and hence the whole system behaves if certain components enter an undefined state or simply an unexpected state. While this methodology would give an indication how models and the system reacts it is not possible to trace how errors propagate inside the system. Getting this information requires instrumentation of TLM ports and exports and track faulty transactions throughout the simulated system. That topic is being investigated by Wagner in his PhD thesis [151].

In the software industry Continuous Integration becomes more and more popular and permeates an increasing number of application domains. Coming originally from web application development it has made its way into mobile application and now the methodology finds its way into embedded system design and testing. This advance is made possible through the usage of standard software-development technologies like distributed version-control systems and pure software solutions for simulating and testing hardware designs.

In the past, every design had to be verified using RTL simulations which take time and offer no flexibility. When simulation models are developed in a shared, version-controlled environment, it is beneficial to also include automatic testing of submitted code.

When automated tests are established for code changes they can also be used to enable commit gating. A developer should not be able to submit code that breaks the simulation model or alters its functionality in such a way that it does not correspond any more to the specification. Automated tests can included simple build checks to see if the code compiles at all or static code analysis to verify adherence to the coding standards.

With the introduction of UVM and snapshotting into SystemC simulations, as I have implemented in this thesis, it is possible to run complicated test setups very quickly. The snapshots ensure a fixed internal state of the simulation models and can be used to ensure constant behaviour through all development phases.

A CI setup, combined with many slave machines to run tests on, can enable an organisation to run a large selection of tests simultaneously with each code change. With the proliferation of cloud technologies, a continuous build and test setup can scale as much as the budget allows. Operating such infrastructure in the cloud might be much cheaper than operating own data centres, but they are far from being free.

Such a distributed computing setup with snapshotable SystemC simulations can also be used to run distributed simulations with varying configurations with tools like Slurm [152]. Such tools originally come from the high performance computing domain, but they can also be used to leverage spare computing power of developer workstations for building and testing purposes. Prebuilt binaries can be distributed via network shares and the same shares can be used to collect results of the test runs executed on the individual workstations.

In the space domain, it makes sense to run distributed simulations for varying parameter configurations and to run fault injection analysis with varying injection rates. Distributed systems do not suffer from the same limitations as real hardware fault injection setups. With a distributed set of virtual platforms the effect of faults on a specific embedded system can be analysed much faster and also much earlier in the development cycle.

Of course fault injection is the main concern in the space and aerospace domains, as I have mentioned earlier. However, there are also other application domains that benefits from saving the state of simulation models and being able to run distributed test campaigns.

One such application domain is the *Internet of Things*, where many simple embedded systems are connected via network to transform ordinary environments into smart environments. The most prevalent example for this technology is right now the *smart home*. In [153] Wenninger et al. explore possibilities to use SystemC simulations for smart home applications. They focus more on the modelling of analogue values such as temperature curves, but the system they describe could also benefit from the ability to run simulation of many small embedded systems from the same state and observe the behaviour of the distributed system.

In an earlier work I have used USI to create HDL models from virtual platform configurations [28]. With the ability to load model state into virtual platform simulation the reverse direction becomes interesting as well. Koch et al. describe in [154] how they created a software tool that modifies hardware designs to enable state read-back from a running hardware design in an FPGA. Their initial idea was to use shift register based scan chains to read out the internal state, but shift registers are inherently slow. They require lots of clock cycles to read out all the data. Then they duplicated all the flip-flops in the design to create a shadow copy of the internal state, which can then be read out using scan chains within just one clock cycle. Their goal with this work is to design reliable hardware. They

can use the internal state of the designed modules to run tests or even to simulate fault injection. As I have done during the implementation of my snapshotting framework, they have worked with a very simple state machine to proof their concept.

In the future it might be possible to convert such scan chain based snapshots into text based snapshots that can be used with my snapshotting framework.

9 Summary

This work presents one way towards making SystemC/TLM simulations and thereby virtual platforms more efficient. This is achieved by introducing checkpointing functionality into the well established virtual platform framework SoCRocket. With the introduction of the UVM library for SystemC into the mix, I make virtual platforms ready for automated testing in a CI workflow. This brings together modern embedded-system design-methodologies with current software-development methodologies.

Developers still struggle when it comes to debugging complex systems, not because of a lack of skill, but because the systems sheer complexity has not been tackled yet through appropriate tool support. As I mentioned in the introduction, the International Technology Roadmap for Semiconductors considers tools supporting developers in debugging tasks one of the grand industry challenges. One small step towards solving this challenge is giving developers access to internal states of systems through introspection tools. Introspection techniques enable snapshotting of internal states and thereby greatly improve developer productivity through reduction of execution times or the ability to parallelize certain tasks.

Tasks profiting from parallel execution are plenty in the debugging and analysis corner. One notable example is the exploration of multiple configuration options. These multiple configurations can be explored in parallel combining the results into one dataset. Another example is booting an operating system on a simulated system and subsequently running software tests on it. The simulated system can be restored from a snapshot where the operating system was already booted and various software tests can be run from that state in parallel. With the increasing prevalence of cloud computing, compute resources become a commodity. Efficient use of these distributed resources requires the ability to scale, which can be achieved through portable snapshots. As a side note, portable snapshots are exactly what enables cloud computing, the ability to quickly boot up virtual machines with preloaded applications.

Previous attempts of introducing snapshotting into SystemC/TLM based virtual platforms are described in detail in the state of the art chapter. Suffice it to say here that they were not successful. One snapshotting implementation relied on external tools to do the actual snapshotting and at the same time limited the model developers in their implementation choices. Another used an established user-level snapshotting tool in conjunction with a modified SystemC kernel to achieve its goal. At least this way the model developers retained their freedom of model implementation choices. However, they could switch simulator implementations any more. None of the aforementioned implementations followed the SystemC standards.

From these previous works and further research, I identified a set of requirements for virtual platform snapshotting. The simulation shall be restored *reliably* into the same state as its original. Snapshotting shall be *transparent* to the model developer. The snapshots shall be independent of library or simulation kernel versions to achieve *portability*. Snapshot files shall be stored in a format which can be inspected and *modified* by developers without any special tools. The snapshotting and restoration processes should not have

a significant negative impact on simulation *performance*. The snapshotting implementation shall not affect *compatibility* with external tools such as debuggers or profilers. The snapshotting functionality shall be able to handle file pointers and other *operation system resources*. Snapshotting shall be implemented in a *self-reliant* way, in order to not introduce any dependencies to external software. With the information collated in the state of the art Chapter the conclusion has to be that the requirements can only be met by implementing application-level checkpointing.

One challenge that needed to be solved to meet the requirements of modifiable and portable snapshots is the serialization of snapshot data. Serialization of data types is a built-in feature in many modern programming languages. Although, C++ can be considered a modern language, even in its newer version it has not gained built-in serialization support. However, developers can draw from a large selection of libraries implementing serialization for C++. In this work the Cereal library was chosen for its portability and relatively low complexity. The library exists entirely in header files and is written using the C++11 language standard, which already simplifies many constructs that have been much more cumbersome in previous C++ versions.

Another remarkable feature of the Cereal library is its extensibility. The supported data types for serialization were extended through several SystemC specific types, such as signals and registers. Since most models already store their state in registers, this will enable already a large number of existing models for snapshotting. With the evaluation of Gaisler's interrupt controller, I have demonstrated that it is very straightforward to snapshot existing models without the need to touch their code or in this case even extend the serialization data types.

After having solved the serialization problem for the SystemC specific data types, I created a snapshot manager class to manage the snapshotting process. The snapshot manager class achieves four goals. First, it contains macros that allow accessing private members of other classes during the snapshotting process. This way access to any internal state, even from the SystemC kernel, is ensured without the need to modify other classes. Second, it uses the SystemC kernels extended phase callbacks to hook itself into the simulation phases to load the model snapshot data at the right time or to save a snapshot at a specific phase. Third, being implemented as a standard SystemC model allows it to access the full SystemC model hierarchy, which is essential for snapshotting. Lastly, it handles the serialization of the various models and their data types using the Cereal library described above.

Furthermore, I implemented a class following the well established decorator pattern, that can be used to mark models as snapshotable during their instantiation. Accessing internal SystemC kernel member variables is made possible through the aforementioned macros that allow accessing private class members. One such prominent private member variable of the SystemC kernel is the simulation time. The SoCRocket signal implementation needed to be extended to allow side-effect-free reading of signal values in order to serialize them in the snapshotting process.

Integrating simulations in virtual platform into a Continuous Integration workflow requires very short turnaround times for the simulation. Usually, CI is used to check if any small code changes break functionality of the system and if they can be integrated properly. To verify this as well as prove that the short turnaround times requirement is met, tests that produce meaningful results in a very short time are needed. This directly relates to the

stated requirements for performance and reliability.

Using snapshots during debugging is quite common, although it is not yet fully established in validation and testing. Having the ability to preload a virtual platform with a specific state, even after recompiling certain models can enable very fast test execution and thereby enable the use of virtual platforms in CI.

In the embedded system domain, UVM is an established methodology for validation testing. With the arrival of the UVM SystemC library, this very powerful testing methodology and framework became available to SystemC-based virtual platforms. Using UVM enables test driven development for embedded systems using virtual platforms. When the virtual platform has snapshotting support the test execution times can be drastically reduced, by loading the desired model states directly before execution of a test session.

The possibility to run distributed test campaigns opens itself up when the snapshots portability requirement is met. JSON, a very common data exchange format, is used as format to store the snapshot data. The snapshot data itself does not contain any machine specific information. All snapshot data is directly related to the virtual platform and the models it comprises. Therefore, it is possible to distribute a snapshot over a set of machines and explore different execution paths on different machines. Alternatively, the same test could be run with a diverse set of snapshots as starting point.

The evaluation shows that the requirements formulated at the beginning could be met. In the evaluation chapter, I compared my snapshotting implementation against the user-level snapshotting tool *DMTCP* as no other application-level SystemC snapshotting implementation was publicly available at this point. The following metrics were used for the comparison: overhead, performance reduction, latency and checkpoint size. Overhead is meant here not in the time sense, but in the sense of added code complexity through inclusion of the snapshotting implementation. Performance reduction is the time penalty introduced to a more complex code base. The latency measures the time spent on the actual snapshotting tasks. Checkpoint size is self-explanatory.

All these metrics were evaluated using a very basic SystemC model. For the overhead metric the winner is my application-level snapshotting implementation. The same goes for performance and latency measurements. Here, my snapshotting framework won because of its very lean implementation without the need for any extra threads or processes. The compilation times between my implementation and a SystemC virtual platform plus *DMTCP* were similar due to the header-only implementation of the Cereal serialization library. The headers need to be included with each recompilation which adds up in the end. For checkpoint size the winner was again very clear. My implementation uses the text-based JSON format, whereas *DMTCP* utilizes a custom binary format. This custom binary format does not meet the requirements of portable and modifiable snapshot files. With *dmtcp* it would not be easy to implement a CI workflow as I have detailed it above.

After these synthetic benchmarks the next step was to look at a more realistic use case. Gaisler's interrupt controller was selected as DUT for a UVM test bench with snapshotting enabled. The previous benchmarks already included a UVM-enabled SoCRocket version. During the adaption of the UVM test bench code it was apparent how easy it is to adapt the code to any new DUT. With a little redesign, using the factory pattern, the test bench implementation can be made universal.

Besides the quantitative analysis of my work in the evaluation I also considered the

qualitative aspects of my implementation. It was possible to leave the SystemC kernel untouched, through employing some elaborate C++ workarounds, that were needed to overcome language constraints. The snapshotting framework presented in this thesis is built on the SystemC standards. Model developers can use standard-compliant templates and model code generators, with only slight modifications, to create new models with snapshotting functionality already built in.

The evaluation proved the high reusability of the UVM test bench code presented in this thesis. The UVM SystemC library used in this thesis is an early release, which can be further improved with subsequent releases. Resulting in better test bench code, which can be implemented in a fully generic manner. Such generic test bench code with snapshotting built in will greatly improve test execution times and ease of test development.

The Cereal library as well as the snapshotting framework presented in this thesis are distributed as C++ headers in order to achieve good portability and easy integration of the code. Aiming for portability and easy integration can lead to higher code overhead and thereby have a negative impact on compilation times. However, optimizing library code by following the factory design pattern is an option to reduce code overhead. Although, effort and benefit have to be balanced here.

My snapshotting approach is ahead of the standardization roadmap of the CCI working group within Accellera. Standardization of the save and restore features for virtual platforms is still far away judging from the time it took to standardize configuration parameters. The SoCRocket framework which I based my implementation on already uses a precursor of the standardized configuration parameters. Furthermore, the register implementation which I used for serialization is based on a proposal by Cadence for a standardized register implementation. Since Cadence has influence on the standardization, the final standardized register implementation is likely to be very close to the current SoCRocket implementation.

This work contributes an extension to the SoCRocket virtual platform framework to enable snapshotting. The snapshotting extension can be considered a reference implementation as the utilization of current SystemC/TLM standards makes it compatible to other frameworkds. Since the snapshotting implementation presented here already uses standardized interfaces for configuration parameters and registers, it is the perfect candidate for standardization of save and restore functionality. Furthermore, integrating the UVM SystemC library into the framework enables test driven development and fast validation of SystemC/TLM models using snapshots. These extensions narrow the design gap by supporting designers, testers and developers to work more efficiently.

10 Appendix

10.1 UVM details

10.1.1 UVM phases

The implementation of a test with UVM is based on the standardized execution of phases. These phases are consistent with the phases of a SystemC simulation. This means for example that a module must not be constructed during the runtime phase of the simulation. UVM stems originally from a SystemVerilog environment, therefore the behaviour had to be adapted somewhat to the new SystemC environment. Pre-existing phases were mapped from SystemC to UVM.

As an example we will look at the UVM `build_phase`. It has been mapped to the SystemC phase `before_end_of_elaboration` [135]. Consequently, the components of the test environment can be constructed in this phase.

Figure 10.1 shows the phases included in the UVM SystemC library. These phases are defined as callback functions in the class `uvm_component` of which all other classes in a test environment are derived. With the help of these callback functions, a developer can implement application code that will be executed in a predefined order. The phases are not limited to the ones shown here. It is possible to extend the library with custom phases.

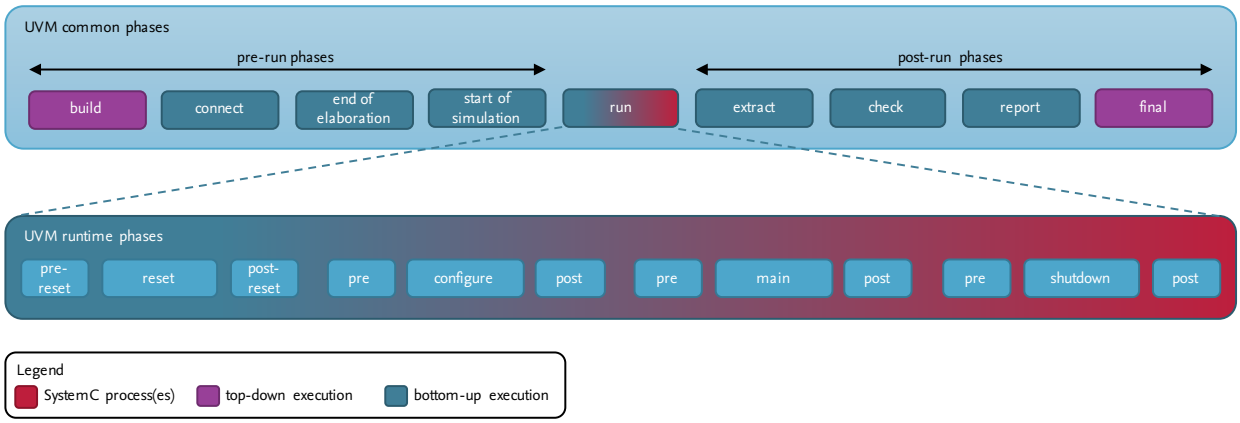

Figure 10.1: UVM phases according to [155]

In principal, the phases are executed sequentially, albeit a phase can also contain several function calls belonging to the involved components which will be executed in parallel. Furthermore, the runtime phases can be executed as concurrent processes [135]. Hence, the flow of a test needs to be controllable to ensure correct execution.

UVM SystemC comes with an objection mechanism to achieve this flow control. The objection mechanism offers hierarchical status communication. Each built-in phase has a so-called objection (class `uvm_objection`). An objection provides status information for the synchronization of a test. The objection indicates if it is safe to end one phase and transition

to the next. The usage of the objection mechanism for a component (or a sequence) is comprised of the following two steps:

raise objection At the start of a procedure that needs to be finished before the end of the current phase, an objection is raised through a function call. This objection refers to the progression into the phase and will be forwarded upwards in the hierarchy.

drop objection When the aforementioned procedure has finished, a function call is used again to drop the objection.

Only if there are no more objections raised in the current phase, the simulation will advance to the next phase.

While the mode of activity of a component determines the use of the objection mechanism, it is not so clear for sequences. Here, we have mainly two options of applying the objection mechanism:

1. **sequence without access to the phase object**

   Since this sequence does not have access to the phase object it is not possible to call the function to raise an objection. Instead the caller of the sequence is responsible for this. After the caller (for example a component) calls the *raise objection* function, it starts the sequence on its sequence instance. When the sequence returns control to the caller, it calls the *drop objection* function. If that was the last objection, the phase is advanced.

2. **sequence with access to the phase object**

   The caller of a sequence passes a reference to the phase in which it (the sequence) is started. This is achieved by assigning a pointer to the current phase to the public member variable `starting_phase`. Afterwards the sequence is started as described above. Here the objection mechanism will only be used, when the variable `starting_phase` was set before the start of the sequence. Where the objection mechanism is placed, is not strictly defined. Good options are, for example, the pairwise sequence callbacks `pre_body`/`post_body` and `pre_start`/`post_start`.

Furthermore, there is an additional method of starting a sequence. The configuration database can be used to store a default sequence for each runtime phase. The sequencer then checks if the configuration database has a resource called `default_sequence` which is assigned to the sequencer. The check takes place at the start of every runtime phase with the name of the phase attached to the access path of the database. The `default_sequence` resource is usually application-specific and can be assigned in one of the higher-level test components. In general, the configuration happens in the component test which is derived from `uvm_test`. When using the `default_sequence` mechanism, a developer can also take advantage of the factory mechanism included in the class library. The configuration database and its combination with the factory will be explained in the next Section.

10.1.2 UVM Factory and Configuration Database

UVM offers through the construct of *configuration database* and *factory* a hierarchy-independent method of component configuration. The stored configuration can be accessed from

anywhere within the test environment. The configuration is not limited to simple variable storage, but also allows exchanging objects that are already residing in it. Moreover, not only object-specific settings but also type specific settings can be made.

The moment during simulation at which an element from the database is accessed does not matter. Access to all data types is allowed during both, elaboration phase and runtime phase of the simulation. Creating data types located in the *factory* is restricted to the elaboration phase of the SystemC simulation. The *factory* implementation of the UVM class library follows the classic factory C++ design pattern described in [117]. Information in the database can either be accessed via name (as string) or via type handle.

Furthermore, an application can decide to limit visibility or accessibility of configuration values. More specifically, a path is passed as string parameter to the configuration value. This path describes at which point in the component hierarchy access to the value is allowed. With the path strings placeholders following the *glob-pattern*- or regular expression syntax are supported as well [135].

The *factory* in UVM is generally used to create objects for specific tests or to replace objects in the database to adapt the environment to a modified test. It is important to keep in mind that only previously registered objects can be created or replaced.

Since *factory* and *configuration database* are quite abstract concepts, I will show some generic code examples.

```
1 // in sc_main
2 uvm_config_db<int>::set(0, "topenv.*", "debug", 1);
3
4 // in module named "topenv"
5 int debug_var;
6 uvm_config_db<int>::get(this, "*", "debug", debug_var);
```

Listing 10.1: Writing and reading of a setting in the *configuration database*

Listing 10.1 shows read and write access to the *configuration database*. The class `uvm_config_db` offers the static methods `set` and `get` with which database modification can be done. In line 2 the integer variable with name “debug” is written. The first two parameters declare the visibility. Since the first parameter is 0 here, only the second one has an effect. The variable *debug* can thus be accessed from all components below *topenv* within the hierarchy. Lines 5 and 6 show how to read the value from the *configuration database*. This read access shall happen within the module *topenv*. Analogous to the `set` method, the first two parameters specify the access context. The `this` pointer gives the *topenv* module. The asterisk means that all components that are below *topenv* in the hierarchy will be accessed.

Listing 10.2 shows usage of the *factory*. Usually the *factory* is used during the `build_phase` to create subcomponents. This use case is demonstrated in line 14. The factory access happens through the static method `create` which is defined in the namespace `type_id`. By calling this method, an object of the type `subcomponent` with the name *mysub* is instantiated at the pointer address. By supplying the `this` pointer as parameter it is possible to have the method create a component path as needed by the *configuration database*. This shows how the *factory* facilitates simple use of the *configuration database*.

```
1  class subcomponent : public uvm_component
2  {
3    UVM_COMPONENT_UTILS(sub);
4    ...
5  };
6
7  class top_env : public uvm_env
8  {
9    subcomponent* sub;
10   ...
11   void build_phase(uvm_phase& phase)
12   {
13     ...
14     sub = subcomponent::type_id::create("mysub", this);
15   }
16 };
```

Listing 10.2: Usage of the *factory*

To be able to create objects with the *factory* this way, the desired object's class needs to contain the necessary infrastructure. This infrastructure is not available by default and has to be supplied. The UVM class library includes convenient macros to add the aforementioned infrastructure to a class.

As mentioned above, it is also possible to let the *factory* exchange already registered types. This can be done in two ways:

type overrides
: UVM SystemC types or derived types from UVM classes come with the function `set_type_override_by_type` which allows to change types of objects being created. The function takes pointers of type `uvm_object_wrapper` as parameters. Alternatively, the function `set_type_override_by_name` can be used which takes object names as parameters. During the creation phase of the object, the factory will now create an object of the new type and not the previously defined one. A new type is required to be derived from the old type. This creates compatibility between object classed by using the inherent polymorphism of C++.

instance overrides
: Analogous to the overriding of general type information described above, it is also possible to exchange types of single instances with the help of the *factory*. For this task, the function `set_inst_override_by_type` can be used. Again, an analogous function that works with names is available. The function takes a path to the instance that shall be changed as well as the old and new types as parameters. The path is viewed relative to the current component.

As you probably have already surmised, the *factory* together with the *configuration database* provide a powerful solution to test bench configuration and customization. Furthermore,

they allow implementing the test environment in a way that makes it reusable for many tests without the need of much modification. Hence, building the perfect base for generalizing test benches.

10.2 Code Listings

```
#! /usr/bin/env python
# -*- coding: utf-8 -*-
# vim: set expandtab:ts=4:sw=4:setfiletype python
import os
from waflib.Errors import ConfigurationError

def options(self):
    self.add_option(
        '--with-cereal',
        type='string',
        help='Basedir of your Cereal installation',
        dest='cerealdir',
        default=os.environ.get("CEREAL_HOME")
    )

def find(self, path = None):
    if path:
      incpath = [os.path.join(os.path.abspath(os.path.expanduser(os.path.
          expandvars(path))), "include")]
    else:
      incpath = []

    self.check_cxx(
      header_name   = "cereal.hpp",
      uselib_store  = 'CEREAL',
      mandatory     = True,
      includes      = incpath,
      use           = '',
      okmsg         = "ok",
      errmsg        = 'Cereal not found please give the location with --
          cereal=',
      fragment      = '''
           #include <cereal/cereal.hpp>

           extern "C" {
               int main(int argc, char** argv) {
                   return 0;
               };
           }
      '''
```

```
39      )
40  #TODO also check for other cereal headers and custom types
41
42  def configure(self):
43      try:
44          if self.options.cerealdir:
45              find(self, self.options.cerealdir)
46          else:
47              find(self)
48      except ConfigurationError as e:
49              name    = "cereal"
50              version = "",
51              self.dep_fetch(
52                  name    = name,
53                  version = version,
54                  git_url = "https://github.com/USCiLab/cereal",
55                  base    = name,
56              )
57              find(self, self.dep_path(name,version).split("-",1)[0])
```

Listing 10.3: Build script for Cereal

10.3 Evaluation Data

Iteration	DMTCP	DMTCP (debug)	`sr_snapshot`
1	339	386	0,481482
2	366	356	0,48821
3	358	391	0,612247
4	334	388	0,335963
5	329	478	0,437083
6	342	367	1,0886
7	339	527	0,728836
8	342	426	0,53395
9	330	358	0,355853
10	340	473	0,42462
Average:	341,9	415	0,5486814

Table 10.1: Snapshot time comparison raw data

Iteration	counter_base	counter_uvm_sm	counter_sm	counter_uvm	dmtcp
1	80286	82925	82025	76101	32720
2	77379	89400	78655	78419	32730
3	76187	84187	79081	78066	32580
4	77815	87241	82646	76277	32380
5	76045	82910	80977	74925	32830
6	76054	82469	81623	75336	32300
7	75102	89489	80983	79855	32590
8	77987	84785	79177	75035	32570
9	75317	85532	79695	78140	32610
10	75802	89557	81336	74602	32250
Average:	76797,4	85849,5	80619,8	76675,6	32556

Table 10.2: Project compile time raw data

Iteration	counter_base	counter_uvm_sm	counter_sm	counter_uvm
1	6783	19828	12855	14090
2	6564	19895	12864	13865
3	8526	19682	12979	13910
4	6653	19737	13127	13970
5	6761	19652	12912	14693
6	6457	19624	12953	13985
7	6564	19719	15053	14022
8	6738	19509	13133	13988
9	6550	19487	12986	13900
10	6511	20188	13074	14503
Average:	6810,7	19732,1	13193,6	14092,6
Average times 4:	27242,8	78928,4	52774,4	56370,4

Table 10.3: `sc_main` recompilation time raw data

Iteration	counter_base	counter_uvm_sm	counter_sm	counter_uvm
1	10731	11472	11364	10865
2	10636	11583	11387	10862
3	10689	11434	11345	10669
4	11966	11449	11336	11043
5	10532	11486	11191	10762
6	10689	11367	11194	10894
7	10437	11319	11242	10833
8	10659	11691	12725	10912
9	10498	11373	12018	10884
10	10504	11463	11492	10905
Average:	10734,1	11463,7	11529,4	10862,9
Average times 8:	85872,8	91709,6	92235,2	86903,2

Table 10.4: SrCount recompilation time raw data


```
{
    "value0": [],
    "simtime": {
        "time": 155390000.0
    },
    "value1": {
        "irqmp.register.level": 7280
    },
    "value2": {
        "irqmp.register.pending": 4294967294
    },
    "value3": {
        "irqmp.register.force": 0
    },
    "value4": {
        "irqmp.register.clear": 0
    },
    "value5": {
        "irqmp.register.mpstat": 537264131
    },
    "value6": {
        "irqmp.register.broadcast": 0
    },
    "value7": {
        "irqmp.register.asymctrl": 0
    },
    "value8": {
        "irqmp.register.irqctrlsel0": 0
    },
```

```
    "value9": {
        "irqmp.register.irqctrlsel1": 0
    },
    "value10": {
        "irqmp.register.mask_0": 1717986918
    },
    "value11": {
        "irqmp.register.force_0": 0
    },
    "value12": {
        "irqmp.register.eir_id_0": 0
    },
    "value13": {
        "irqmp.register.mask_1": 2576980376
    },
    "value14": {
        "irqmp.register.force_1": 0
    },
    "value15": {
        "irqmp.register.eir_id_1": 0
    },
    "value16": {
        "irqmp.rst": true
    },
    "value17": {
        "irqmp.cpu_stat": true
    },
    "value18": {
        "irqmp.irq_ack": 18285065
    },
    "value19": {
        "irqmp.irq_in": {
            "first": 2147483648,
            "second": false
        }
    },
    "value20": {
        "testbench.agent1.driver.object_6": {
            "first": 2147483648,
            "second": false
        }
    },
    "value21": {
        "testbench.agent2.monitor.IRQ_REQUEST": {
            "first": 0,
```

```
            "second": false
        }
    },
    "value22": {
        "testbench.agent2.monitor.port": {
            "first": 10,
            "second": true
        }
    },
    "value23": {
        "testbench.agent2.monitor.port_1": {
            "first": 12,
            "second": true
        }
    }
}
```

Listing 10.4: JSON snapshot file from UVM platform with IRQMP as DUT

Bibliography

[1] K. Lu, D. Müller-Gritschneder, and U. Schlichtmann, "Accurately timed transaction level models for virtual prototyping at high abstraction level," in *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '12, (San Jose, CA, USA), pp. 135–140, EDA Consortium, 2012.

[2] A. Gerstlauer, S. Chakravarty, M. Kathuria, and P. Razaghi, "Abstract System-Level Models for Early Performance and Power Exploration," in *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*, pp. 213–218, Jan 2012.

[3] R. Collett and D. Pyle, "What happens when chip-design complexity outpaces development productivity?," *McKinsey on Semiconductors*, vol. 3, pp. 24–33, 2013.

[4] "International Technology Roadmap for Semiconductors." http://www.itrs.net.

[5] F. Ghenassia, *Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems.* Springer US, 2005.

[6] "Accellera Standards - SystemC." http://accellera.org/downloads/standards/systemc. Accessed: 2018-05-27.

[7] Aeroflex/Gaisler, "Aeroflex/Gaisler IP and manual download." http://www.gaisler.com/index.php/downloads. Accessed: 2016-04-04.

[8] "The Schiaparelli Lesson – Unusual and Faulty Conditions." https://software.intel.com/en-us/blogs/2017/01/11/the-schiaparelli-lesson-unusual-and-faulty-conditions.

[9] Oxford University Press, "Oxford Dictionaries Online." https://en.oxforddictionaries.com, 2017.

[10] J. S. Plank, M. Beck, G. Kingsley, and K. Li, *Libckpt: Transparent checkpointing under unix.* Computer Science Department, 1994.

[11] W. R. Dieter and J. E. Lumpp, "A user-level checkpointing library for posix threads programs," in *Fault-Tolerant Computing, 1999. Digest of Papers. Twenty-Ninth Annual International Symposium on*, pp. 224–227, IEEE, 1999.

[12] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Comput. Surv.*, vol. 34, pp. 375–408, Sept. 2002. check references again for marked sections!

[13] A. Beguelin, E. Seligman, and P. Stephan, "Application level fault tolerance in heterogeneous networks of workstations," *Journal of Parallel and Distributed Computing*, vol. 43, no. 2, pp. 147–155, 1997.

[14] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "KVM: The Linux Virtual Machine Monitor," in *Proceedings of the Linux symposium*, vol. 1, pp. 225–230, 2007.

[15] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield, "Remus: High availability via asynchronous virtual machine replication," in *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pp. 161–174, San Francisco, 2008.

[16] P. Hargrove, J. Duell, and E. Roman, "Berkeley lab checkpoint/restart (BLCR) for Linux clusters," in *Journal of Physics: Conference Series*, vol. 46, p. 494, IOP Publishing, 2006.

[17] R. Gioiosa, J. C. Sancho, S. Jiang, F. Petrini, and K. Davis, "Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers," in *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, p. 9, IEEE Computer Society, 2005.

[18] J. Bonwick, M. Ahrens, V. Henson, M. Maybee, and M. Shellenbaum, "The zettabyte file system," in *Proc. of the 2nd Usenix Conference on File and Storage Technologies*, vol. 215, 2003.

[19] O. Rodeh, J. Bacik, and C. Mason, "BTRFS: The Linux B-tree filesystem," *ACM Transactions on Storage (TOS)*, vol. 9, no. 3, p. 9, 2013.

[20] J. S. Plank, K. Li, and M. A. Puening, "Diskless checkpointing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 10, pp. 972–986, 1998.

[21] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *Proceedings of the 2010 ACM/IEEE international conference for high performance computing, networking, storage and analysis*, pp. 1–11, IEEE Computer Society, 2010.

[22] K. Li, J. F. Naughton, and J. S. Plank, "Low-latency, concurrent checkpointing for parallel programs," *IEEE transactions on Parallel and Distributed Systems*, vol. 5, no. 8, pp. 874–879, 1994.

[23] IEEE-SA Standards Board, "IEEE Standard for Standard SystemC Language Reference Manual," *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2011)*, 09 2011.

[24] "SoCROCKET Virtual Platform - SystemC." http://www.esa.int/Our_Activities/Space_Engineering_Technology/Microelectronics/SoCROCKET_Virtual_Platform_-_SystemC. Accessed: 2017-07-24.

[25] E. CONSORTIUM, "Embedded multi-core systems for mixed criticality applications in dynamic and changeable real-time environments." http://www.artemis-emc2.eu. Accessed: 2017-07-24.

[26] SoCRocket Team, "SoCRocket sources." https://github.com/socrocket. Accessed: 2017-07-24.

[27] R. Meyer, J. Wagner, R. Buchty, and M. Berekovic, "Universal scripting interface for systemc," in *DVCon Europe Conference Proceedings 2015*, Nov 2015.

[28] B. Farkas, S. A. A. Shah, J. Wagner, R. Meyer, R. Buchty, and M. Berekovic, "An open and flexible systemc to vhdl workflow for rapid prototyping," in *Design and Verification Conference (DVCon) Europe 2016, October 19 - 20, 2016 Munich, Germany*, Oct 2016.

[29] R. Meyer, J. Wagner, B. Farkas, S. Horsinka, P. Siegl, R. Buchty, and M. Berekovic, "A Scriptable Standard-Compliant Reporting and Logging Framework for SystemC," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, no. 1, 2016.

[30] R. Meyer, B. Farkas, S. A. A. Shah, and M. Berekovic, "Transparent SystemC Model Factory for Scripting Languages," in *DVCon USA Conference Proceedings 2017*, 2017.

[31] S. A. A. Shah, S. A. Horsinka, B. Farkas, R. Meyer, and M. Berekovic, "Automatic Exploration of Hardware/Software Partitioning," in *DVCon USA Conference Proceedings 2017*, 2017.

[32] T. Schuster, *SoCRocket: eine flexible erweiterbare Virtuelle Plattform zum Entwurf robuster Eingebetteter Systeme.* PhD thesis, TU Braunschweig, 2015.

[33] B. Farkas and R. Meyer, "Socrocket 101," in *AHS - NASA/ESA Conference on Adaptive Hardware and Systems*, 2015.

[34] J. A. M. Berrada, "SystemPython: a Python extension to control SystemC SoC simulations," *GreenSocs meeting presentation, Design and Test in Europe Conference (DATE)*, 04 2007.

[35] "Simplified Wrapper Interface Generator (SWIG)." http://www.swig.org, 2014.

[36] S. Swan and J. Cornet, "Beyond TLM 2.0: New Virtual Platform Standards Proposals from ST and Cadence," in *NASCUG at DAC*, 2012.

[37] K. Beck, *Extreme Programming Explained. Embrace Change.* Addison Wesley, 1st edition ed., 2000.

[38] M. Fowler, "Continuous Integration." http://martinfowler.com/articles/continuousIntegration.html. Accessed: 14.12.2016.

[39] J. Engblom, "Virtual to the (near) end-using virtual platforms for continuous integration," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE, 2015.

[40] "IEEE Standard for Universal Verification Methodology Language Reference Manual," *IEEE Std. 1800.2-2017*, 2017.

[41] P. McLellan, "UVM Is Now IEEE 1800.2 and There's a Ten-Year Story to That." https://community.cadence.com/cadence_blogs_8/b/breakfast-bytes/archive/2017/05/04/stan-dards, May 2017. Accessed: 2017-07-04.

[42] Accellera Systems Initiative, *Universal Verification Methodology (UVM) 1.2 User Guide*, Oct 2015.

[43] K. M. Chandy, "A survey of analytic models of rollback and recovery stratergies," *Computer*, vol. 8, no. 5, pp. 40–47, 1975.

[44] B. Randell, "System structure for software fault tolerance," *IEEE Transactions on Software Engineering*, vol. SE-1, pp. 220–232, June 1975.

[45] R. Koo and S. Toueg, "Checkpointing and rollback-recovery for distributed systems," *IEEE Transactions on software Engineering*, no. 1, pp. 23–31, 1987.

[46] S. Israel and D. Morris, "A non-intrusive checkpointing protocol," in *Computers and Communications, 1989. Conference Proceedings., Eighth Annual International Phoenix Conference on*, pp. 413–421, IEEE, 1989.

[47] P.-J. Leu and B. Bhargava, "A model for concurrent checkpointing and recovery using transactions," in *Distributed Computing Systems, 1989., 9th International Conference on*, pp. 423–430, IEEE, 1989.

[48] C. Hernandez and J. Abella, "Low-cost checkpointing in automotive safety-relevant systems," in *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 91–96, IEEE, 2015.

[49] SoCLib Consortium and others, "The SoCLib project: An integrated system-on-chip modelling and simulation platform," tech. rep., LIP6, 2003.

[50] C. A. Vick and L. G. Votta, "Method and apparatus for computer system diagnostics using safepoints," Aug. 26 2008. US Patent 7,418,630.

[51] C. A. Vick, M. H. Paleczny, J. R. Freeman, and L. G. Votta Jr, "Method for automatic checkpoint of system and application software," Apr. 7 2009. US Patent 7,516,361.

[52] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *ACM SIGOPS operating systems review*, vol. 37, pp. 164–177, ACM, 2003.

[53] M. Rosenblum and M. Varadarajan, *Simos: A fast operating system simulation environment*. Computer Systems Laboratory, Stanford University, 1994.

[54] B. Cully and A. Warfield, "Virtual machine checkpointing," *Xen summit*, 2007.

[55] P. Ta-Shma, G. Laden, M. Ben-Yehuda, and M. Factor, "Virtual machine time travel using continuous data protection and checkpointing," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 1, pp. 127–134, 2008.

[56] H. Liu, H. Jin, and X. Liao, "Optimize performance of virtual machine checkpointing via memory exclusion," in *2009 Fourth ChinaGrid Annual Conference*, pp. 199–204, IEEE, 2009.

[57] E. Park, B. Egger, and J. Lee, "Fast and space-efficient virtual machine checkpointing," in *ACM SIGPLAN Notices*, vol. 46, pp. 75–86, ACM, 2011.

[58] V. Siripoonya and K. Chanchio, "Thread-based live checkpointing of virtual machines," in *Network Computing and Applications (NCA), 2011 10th IEEE International Symposium on*, pp. 155–162, IEEE, 2011.

[59] F. Pérez and B. E. Granger, "Ipython: a system for interactive scientific computing," *Computing in Science & Engineering*, vol. 9, no. 3, 2007.

[60] R. Garg, K. Sodha, Z. Jin, and G. Cooperman, "Checkpoint-restart for a network of virtual machines," in *2013 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 1–8, IEEE, 2013.

[61] F. Bellard, "Qemu, a fast and portable dynamic translator.," in *USENIX Annual Technical Conference, FREENIX Track*, pp. 41–46, 2005.

[62] A. Sari and M. Psarakis, "Checkpointing virtualized mixed-critical embedded systems," in *1st International Workshop on Resiliency in Embedded Electronic Systems*, 2015.

[63] H. M. Le, V. Herdt, D. Große, and R. Drechsler, "Towards formal verification of real-world systemc tlm peripheral models: a case study," in *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*, pp. 1160–1163, EDA Consortium, 2016.

[64] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny, "Checkpoint and migration of unix processes in the condor distributed processing system," tech. rep., Technical Report, 1997.

[65] J. Ansel, K. Arya, and G. Cooperman, "DMTCP: Transparent checkpointing for cluster computations and the desktop," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pp. 1–12, IEEE, 2009.

[66] M. Rieker, J. Ansel, and G. Cooperman, "Transparent user-level checkpointing for the native posix thread library for linux.," in *PDPTA*, vol. 6, pp. 492–498, 2006.

[67] "Unreal Engine." https://www.unrealengine.com.

[68] "Unity Game Engine." https://unity3d.com.

[69] "RetroArch Crossplatform architecture." https://www.libretro.com/index.php/api/, 2014.

[70] "Minecraft." https://minecraft.net.

[71] "Computer Built in Minecraft Has RAM, Performs Division." http://www.escapistmagazine.com/news/view/109385-Computer-Built-in-Minecraft-Has-RAM-Performs-Division. Accessed: 2017-08-08.

[72] Sun Microsystems, "XDR: external data representation standard," RFC 1014, Internet Engineering Task Force, June 1987.

[73] R. Xiaoguang, X. Xinhai, T. Yuhua, F. Xudong, and S. Ye, "An application-level synchronous checkpoint-recover method for parallel cfd simulation," in *Computational Science and Engineering (CSE), 2013 IEEE 16th International Conference on*, pp. 58–65, IEEE, 2013.

[74] "Vim - the ubiquitous text editor." http://www.vim.org.

[75] "GNU Emacs." https://www.gnu.org/software/emacs/.

[76] "unexelf.c - Emacs source repository." http://git.savannah.gnu.org/cgit/emacs.git/tree/src/unexelf.c?id=1a50945fa4c666ae2ab5cd9419d23ad063ea1249. Accessed: 2017-03-02.

[77] "Removing support for Emacs unexec from Glibc [LWN.net]." https://lwn.net/Articles/673724/. Accessed: 2017-03-02.

[78] "The Emacs dumper dispute [LWN.net]." https://lwn.net/Articles/707615/. Accessed: 2017-03-02.

[79] "Preview/ portable dumper [LWN.net]." https://lwn.net/Articles/707619/. Accessed: 2017-03-02.

[80] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[81] M. Slee, A. Agarwal, and M. Kwiatkowski, "Thrift: Scalable Cross-Language Services Implementation," tech. rep., Facebook, 2007.

[82] "Google Protocol Buffers." https://developers.google.com/protocol-buffers/docs/overview.

[83] "boost Serialization." http://www.boost.org/doc/libs/1_64_0/libs/serialization/doc/index.html.

[84] M. Montón, *Checkpointing for virtual platforms and SystemC-TLM-2.0*. PhD thesis, Universitat Autònoma de Barcelona, 2011.

[85] "MessagePack." http://msgpack.org.

[86] W. S. Grant and R. Voorhies, "cereal - A C++11 library for serialization." http://uscilab.github.io/cereal/, 2017.

[87] "Apache Avro™." https://avro.apache.org.

[88] L. E. Lwakatare, T. Karvonen, T. Sauvola, P. Kuvaja, H. H. Olsson, J. Bosch, and M. Oivo, "Towards devops in the embedded systems domain: Why is it so hard?," in *System Sciences (HICSS), 2016 49th Hawaii International Conference on*, pp. 5437–5446, IEEE, 2016.

[89] S. Chessin, "Injecting errors for fun and profit," *Queue*, vol. 8, no. 8, p. 10, 2010.

[90] C. Sauer and H.-P. Loeb, "A lightweight infrastructure for the dynamic creation and configuration of virtual platforms," in *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2015 International Conference on*, pp. 372–377, IEEE, 2015.

[91] P. S. Magnusson, "The virtual test lab," *Computer*, vol. 38, no. 5, pp. 95–97, 2005.

[92] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *Computer*, vol. 35, no. 2, pp. 50–58, 2002.

[93] J. Engblom, "Transporting bugs with checkpoints," in *Proceedings of the System, Software, SoC and Silicon Debug Conference (S4D 2010)*, pp. 15–16, 2010.

[94] T. Yu, W. Srisa-an, and G. Rothermel, "Simtester: a controllable and observable testing framework for embedded systems," in *ACM SIGPLAN Notices*, vol. 47, pp. 51–62, ACM, 2012.

[95] J. Wenninger, M. Damm, J. Moreno, J. Haase, and C. Grimm, "Multilevel sensor node simulation within a tlm-like network simulation framework," in *Architecture of Computing Systems (ARCS), 2010 23rd International Conference on*, pp. 1–6, VDE, 2010.

[96] K. Tokuno and S. Yamada, "Codesign-oriented performability modeling for hardware-software systems," *IEEE Transactions on Reliability*, vol. 60, no. 1, pp. 171–179, 2011.

[97] R. Drechsler, C. Chevallaz, F. Fummi, A. J. Hu, R. Morad, F. Schirrmeister, and A. Goryachev, "Future soc verification methodology: Uvm evolution or revolution?," in *Proceedings of the conference on Design, Automation & Test in Europe*, p. 372, European Design and Automation Association, 2014.

[98] IEEE, *IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows*, ieee std 1685–2014 (revision of ieee std 1685–2009) ed., 2014.

[99] M. Barnasconi, M. Dietrich, K. Einwich, T. Vörtler, J.-P. Chaput, M.-M. Louërat, F. Pêcheux, Z. Wang, P. Cuenot, I. Neumann, *et al.*, "Uvm-systemc-ams framework for system-level verification and validation of automotive use cases," *IEEE Design & Test*, vol. 32, no. 6, pp. 76–86, 2015.

[100] M. Barnasconi, "Building a Coherent ESL Design and Verification Eco-system with SystemC, TLM, UVM-SystemC, and CCI," in *DVCon Europe Conference Proceedings 2016*, 2016.

[101] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson, "GOOFI: Generic Object-Oriented Fault Injection Tool," in *Dependable Systems and Networks, 2001. DSN 2001. International Conference on*, pp. 83–88, IEEE, 2001.

[102] G. Frazier, "How to Save OS Boot Time In Your SystemC Virtual Platform With Save and Restore." https://community.cadence.com/cadence_blogs_8/b/sd/archive/2009/02/18/how-to-save-os-boot-time-in-your-systemc-virtual-platform-with-save-and-restore, February 2009.

[103] W. Klingauf and M. Geffken, "Design structure analysis and transaction recording in systemc designs: A minimal-intrusive approach," in *Proc. FDL*, 2006.

[104] S. Kraemer, R. Leupers, D. Petras, and T. Philipp, "A checkpoint/restore framework for systemc-based virtual platforms," in *System-on-Chip, 2009. SOC 2009. International Symposium on*, pp. 161–167, Oct 2009.

[105] E. Roman, "A survey of checkpoint/restart implementations," tech. rep., Lawrence Berkeley National Laboratory, Tech, 2002.

[106] T. Philipp, D. Petras, and T. Michiels, "Simulation control techniques." US PATENT 2013/0191346, July 2013.

[107] "Cadence SystemC Checkpointing – Observations from Uppsala." http://jakob.engbloms.se/archives/817. Accessed: 2017-08-03.

[108] "SystemC Save and Restore Part 2 - Advanced Usage." https://community.cadence.com/cadence_blogs_8/b/sd/archive/2009/03/09/systemc-save-and-restore-part-2-advanced-usage. Accessed: 2017-08-03.

[109] J. Engblom, M. Montón, and M. Burton, "Checkpointing SystemC Models," in *FDL 2009*, 2009.

[110] Configuration, Control & Inspection (CCI) Working Group Open SystemC Initiative, *Requirements Specification for Configuration Interfaces*, 2009.

[111] T. Wieman, "Deploying SystemC for Complex System Prototyping and Validation," in *DVCon Conference Proceedings 2013*, 2013.

[112] G. Delbergue and T. Wieman, "SystemC Configuration A preview of the draft standard," in *DVCon Europe Conference Proceedings 2016*, 2016.

[113] B. Neifert, "Debugging arm software with 100s of mips and 100Carbon Design Newsletter, March 2011.

[114] B. Neifert and R. Kaye, "High Performance or Cycle Accuracy? You can have both," tech. rep., ARM, 2012.

[115] ARM, *ESL API Developer's Guide*, version 9.1.0 ed., 2017.

[116] G. Xu, "Project Report: Checkpointing on SystemC," tech. rep., Carnegie Mellon University, 2012.

[117] R. Johnson, E. Gamma, R. Helm, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[118] B. Dawes, D. Abrahams, R. Rivera, *et al.*, "Boost C++ libraries." http://www.boost.org.

[119] B.-A. Tabacaru, M. Chaari, W. Ecker, T. Kruse, and C. Novello, "Efficient checkpointing-based safety-verification flow using compiled-code simulation," in *Digital System Design (DSD), 2016 Euromicro Conference on*, pp. 364–371, IEEE, 2016.

[120] T. Becker, "On the tension between object-oriented and generic programming in c++ and what type erasure can do about it." http://www.artima.com/cppsource/type_erasure.html, 2007.

[121] K. Sorokin, "Benchmark comparing various data serialization libraries." https://github.com/thekvs/cpp-serializers. Accessed: 2017-03-02.

[122] R. Görgen and P. A. Hartmann, "Moving SystemC to a New C++ Standard," in *DVCon Europe Conference Proceedings 2016*, 2016.

[123] J. Schaub, "Access to private members: Safer nastiness.." http://bloglitb.blogspot.de/2011/12/access-to-private-members-safer.html, 2011. Accessed: 2017-10-01.

[124] J. Aynsley, "New Features of IEEE Std 1666-2011 SystemC," in *DVCon Conference Proceedings 2012*, 2012.

[125] R. Meyer, "Python SystemC implementation of USI." https://github.com/socrocket/pysc, 2017.

[126] "boost Serialization Tutorial." http://www.boost.org/doc/libs/1_42_0/libs/serialization/doc/tutorial.html.

[127] J. O. Coplien, "Curiously recurring template patterns," *C++ Report*, vol. 7, no. 2, pp. 24–27, 1995.

[128] "Proposed interfaces for interrupt modeling, register introspection and modeling of memory maps from stmicroelectronics, arm, and cadence 1.1." http://forums.accellera.org/files/file/102-proposed-interfaces-for-interrupt-modeling-register-introspection-and-modeling-of-memory-maps-from-stmicroelectronics-arm-and-cadence/. Accessed: 2017-07-25.

[129] "GreenLib." https://www.greensocs.com/get-started#greenlib. Accessed: 2017-07-25.

[130] "SoCRocket Register Implementation." https://github.com/socrocket/sr_register.

[131] "SoCRocket TLM Signal." https://github.com/socrocket/sr_signal.

[132] M. Yip, "RapidJSON - A fast JSON parser/generator for C++ with both SAX/DOM style API." http://rapidjson.org/, 2015.

[133] "cereal::construct<T> Class Template Reference." http://uscilab.github.io/cereal/assets/doxygen/classcereal_1_1construct.html#details.

[134] J. Aynsley, *OSCI TLM-2.0 language reference manual.* Open SystemC Initiative, ja32 ed., jul. 2009.

[135] Accellera Systems Initiative, *UVM-SystemC Language Reference Manual*, DRAFT ed., Dec 2015.

[136] "DMTCP: Distributed MultiThreaded CheckPointing." https://github.com/dmtcp/dmtcp/releases/tag/2.5.2. Accessed: 2018-01-06.

[137] "cloc - Count Lines of Code." https://github.com/AlDanial/cloc. Accessed: 2018-01-06.

[138] J. Gaisler, E. Catovic, M. Isomaki, K. Glembo, and S. Habinc, *GRLIB IP Core User's Manual*. Gaisler Research, 2007.

[139] Accellera, "Accellera working group for Configuration, Control and Inspection," *Website http://www.accellera.org/activities/committees/systemc-cci/*, 2016.

[140] P. A. Hartmann, "Accellera Update: What's New and Cooking in SystemC Standardization," in *DVCon Europe Conference Proceedings 2017*, 2017.

[141] *SystemC Configuration, Control and Inspection Standard*. Accellera, 2018.

[142] "SystemC Evolution Day 2017." http://accellera.org/news/events/systemc-evolution-day-2017.

[143] J. Engblom, H. Zeffer, E. Nilsson, and P. Hartmann, "Checkpointing and SystemC – How Can We Make Them Meet?," in *SystemC Evolution Day 2017*, 2017.

[144] "Cadence Virtual System Platform." https://www.cadence.com/content/cadence-www/global/en_US/home/tools/system-design-and-verification/software-driven-verification/virtual-system-platform.html. Accessed: 2018-01-17.

[145] "Creating SystemC TLM-2.0 Peripheral Models." https://community.cadence.com/cadence_blogs_8/b/sd/posts/creating-systemc-tlm-2-0-peripheral-models. Accessed: 2018-01-17.

[146] "CppHeaderParser 2.7.4." https://pypi.python.org/pypi/CppHeaderParser/. Accessed: 2018-02-11.

[147] R. Meyer, *USI: Universal Scripting Interface for SystemC*. manuscript, TU Braunschweig, 2019.

[148] M. Prior, "IMAGE Failure Review Board Final Report," tech. rep., NASA, Goddard Space Flight Center, 2006.

[149] N. Wang, J. Quek, T. Rafacz, and S. Patel, "Characterizing the effects of transient faults on a high-performance processor pipeline," in *Dependable Systems and Networks, 2004 International Conference on*, pp. 61–70, June 2004.

[150] S. Michalak, A. DuBois, C. Storlie, H. Quinn, W. Rust, D. DuBois, D. Modl, A. Manuzzato, and S. Blanchard, "Assessment of the Impact of Cosmic-Ray-Induced Neutrons on Hardware in the Roadrunner Supercomputer," *Device and Materials Reliability, IEEE Transactions on*, vol. 12, pp. 445–454, June 2012.

[151] J. Wagner, *tbd*. manuscript, TU Braunschweig, 2019.

[152] A. B. Yoo, M. A. Jette, and M. Grondona, "Slurm: Simple linux utility for resource management," in *Workshop on Job Scheduling Strategies for Parallel Processing*, pp. 44–60, Springer, 2003.

[153] J. Wenninger and J. Haase, "Efficient building automation simulation using system on chip simulation techniques," in *Architecture of Computing Systems (ARCS), Proceedings of 2013 26th International Conference on*, pp. 1–6, VDE, 2013.

[154] D. Koch, C. Haubelt, and J. Teich, "Efficient hardware checkpointing: Concepts, overhead analysis, and implementation," in *Proceedings of the 2007 ACM/SIGDA 15th International Symposium on Field Programmable Gate Arrays*, FPGA '07, (New York, NY, USA), pp. 188–196, ACM, February 2007.

[155] S. Schulz, T. Vörtler, and M. Barnasconi, "UVM goes Universal - Introducing UVM in SystemC," in *DVCon Europe Conference Proceedings 2015*, 2015.